Departamento de Ingeniería de Sistemas y Automática

PROYECTO FIN DE CARRERA

Desarrollo de una aplicación para el control directo de las articulaciones del robot humanoide RH-2

Autor: Raúl Morales Tejero

Tutor: Santiago Martínez de la Casa Díaz

Leganés, octubre de 2011

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

Este proyecto que aquí se presenta consiste en la realización de una aplicación para el control directo de las múltiples articulaciones del robot humanoide RH-2 y la generación de movimiento de las mismas. Este robot se encuentra actualmente en fase de desarrollo y pertenece al Proyecto de Robots Humanoides RH del departamento de Ingeniería de Sistemas y Automática de la Universidad Carlos III de Madrid

Para ello, el control directo con los drivers del humanoide se ha desarrollado en un lenguaje de programación C para un sistema operativo Linux.

La comunicación con los módulos de las diferentes articulaciones del robot se ha realizado por medio del protocolo de comunicación CANOpen debido a su fiabilidad y su rapidez, dicho protocolo esta implementado sobre el bus de campo CAN.

La interacción con el usuario se realiza a través de una interfaz diseñada en Matlab que permite el uso de aplicaciones en tiempo real. Esta interfaz tiene diferentes opciones que el usuario puede usar según corresponda.

Para el desarrollo de este proyecto, todas las pruebas han sido realizadas en la estructura mecánica del tobillo RH-2, debido a que dicho robot no está desarrollado completamente.

ABSTRACT

This project consists of an application programming for the direct motor control of the multiple joints of the humanoid robot RH-2 and the generation of movement. This robot is currently under development and it belongs to the humanoid robots research field (RH) of the Department of Systems Engineering and Automation of the University Carlos III of Madrid.

To do this, the application for direct control has been developed using C programming language under Linux operating system.

The communication between software modules of every joint of the robot has been implemented using the CANOpen protocol. Its use provides reliability and high transmission speed. The protocol is implemented on the CAN field bus.

The interaction with the user has been developed with the GUI tools available in Matlab. It allows the development of real-time applications. This interface gives the user different options for control and information.

The results of this project have been tested by means of the use of the mechanical structure of the RH-2ankle because the whole robot is not fully assembled yet.

CAPITULO 1: INTRODUCCIÓN

1.1 Introducción

El proyecto consiste en la realización de una aplicación para el control directo de las múltiples articulaciones del robot humanoide RH-2 y la generación de movimiento de las mismas.

El robot RH-2, aún en fase de desarrollo, pertenece al proyecto de robots humanoides RH del ROBOTICSLAB del Departamento de Ingeniería de Sistemas y Automática de la Universidad Carlos III de Madrid y supone una mejora tanto en el hardware como en el software con respecto a sus predecesores: el RH-0 y el RH-1.

Consiste en un sistema mecánico de 26 (24+2 de la cabeza) grados de libertad, dispuestos de modo que le dotan de apariencia humana tanto en el aspecto como en la capacidad de locomoción y manipulación.

El proyecto se ha desarrollado en el lenguaje de programación C y la comunicación con los módulos ha sido por medio del protocolo de comunicación CANOpen implementado sobre el bus de campo CAN. Este protocolo consigue una comunicación serie para el intercambio de información entre las unidades del sistema y soporta un eficiente control en tiempo real con un nivel de seguridad muy elevado.

La interacción con el usuario se realiza por medio de una interfaz en Matlab que permite el uso de aplicaciones en tiempo real.

Se ha llevado a cabo la puesta en marcha en el laboratorio de la universidad de una estructura mecánica de dos GDL semejante al tobillo del robot humanoide para realizar los ensayos y demostraciones convenientes para verificar el correcto funcionamiento.

1.2 Antecedentes

El Proyecto de Robots Humanoides RH consta con un prototipo ya construido (RH-1) y uno en fase de desarrollo (RH-2). En este apartado se pretende dar una visión general de los aspectos más importantes del diseño de estos robots.

1.2.1 El RH-1.

Este robot se definió como un sistema mecánico de 21 grados de libertad (GDL) todos ellos activos. En cada GDL hay un motor encargado de su movimiento y están dispuestos de modo que le dotan de apariencia humana tanto en el aspecto como en la capacidad de locomoción, con las limitaciones propias de la extrema complejidad del sistema locomotor de un ser humano. El RH-1 fue el resultado del rediseño de algunos elementos hardware de su predecesor RH-0 (conservando los grados de libertad). Ambos fueron desarrollados tomando como modelos los prototipos más avanzados existentes en aquel momento: el ASIMO de Honda y el HRP-2P de Kawada. En la figura 1 se muestran el RH-1, ASIMO Y HRP-2P en este orden.

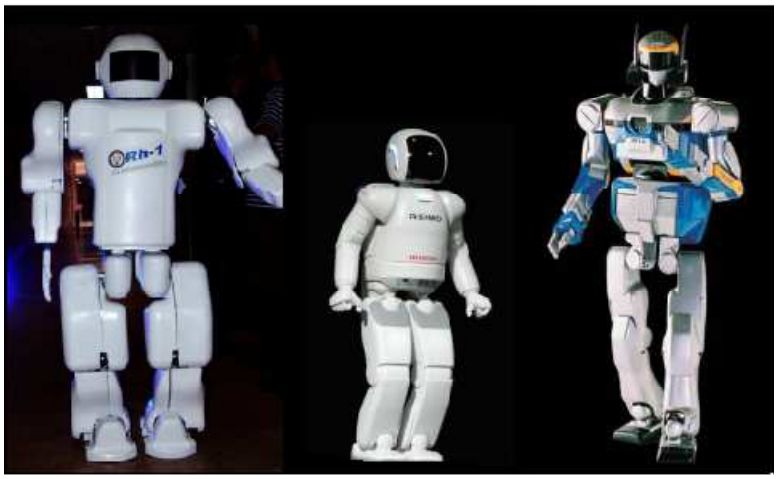


Figura 1: RH1 (izquierda), ASIMO (centro) y HRP-2P (derecha)

El resultado final de diferentes estudios y configuraciones fue la distribución de sus 21 grados de libertad como se muestra en la tabla 1, la verdadera forma del RH-1 se pude observar en la figura 2.

Tabla 1: Grados libertad RH1

GDL	Número	Eje de movimiento
PIERNAS	12 GDL	
Cadera	3 (x2)	Sagital Frontal Transversal
Rodilla	1 (x2)	Sagital
Tobillo	2 (x2)	Sagital Frontal
BRAZOS	8 GDL	
Hombros	2 (x2)	Sagital Frontal
Codo	1 (x2)	Sagital
Muñeca	1 (x2)	Transversal
TRONCO	1 GDL	
Tronco	1	Transversal
CABEZA	2 GDL	
Cámara	2	Trasversal (izquierda-derecha) Sagital(arriba-abajo)
TOTAL	23 GDL con cabeza 21 GDL sin cabeza	

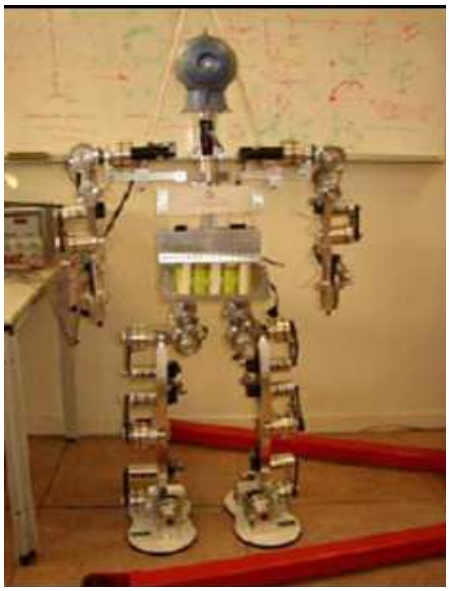
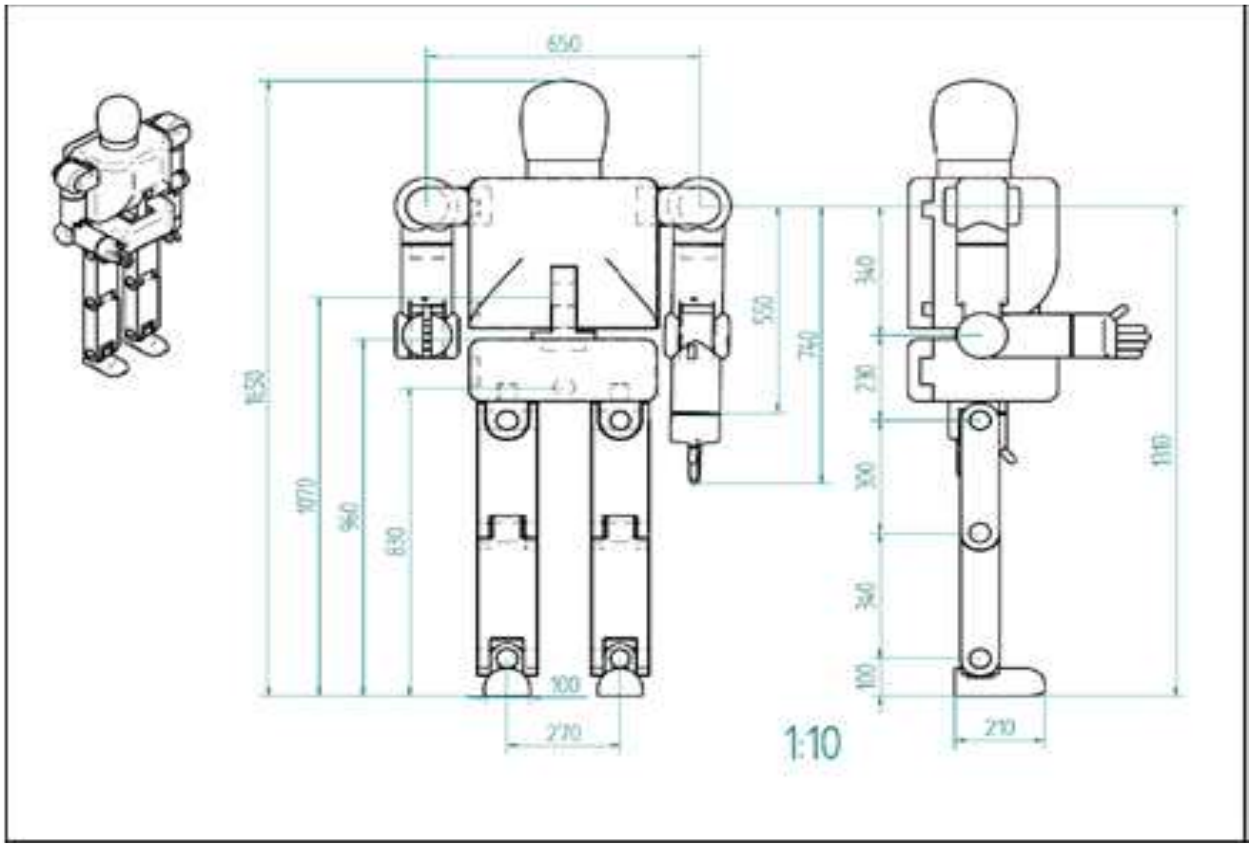


Figura 2: RH1

1.2.2 El RH-2

Para este nuevo robot se estima un peso de 55 Kg y una velocidad de 0.75 m/s durante la caminata. Se calcula que podrá transportar objetos de hasta 2 Kg de peso por brazo e incluso sentarse. Su altura aumenta de 1.2 m a 1.65 m, dotando al robot de un tamaño más acorde al de un humano. Estas medidas se pueden observar en la figura 3.

Figura 3: Medidas prototipo RH2



Para el diseño de este robot se ha buscado una solución para los problemas derivados de las anteriores versiones y teniéndolos en cuenta se ha diseñado un nuevo sistema actualizado acorde a las nuevas tecnologías disponibles. Los anteriores modelos disponían de 21 grados de libertad sin contar con la cabeza.

El RH-2 pretende incluir tres grados más de libertad: uno en cada codo en el eje transversal que permita a los brazos realizar movimientos más parecidos a los humanos y otro grado de libertad en el tronco en su eje sagital para poder controlar de manera más rápida el balanceo del cuerpo hacia delante y atrás y lograr mantener su centro de masa centrado. Estos datos se pueden observar en la tabla 2 y su representación en la figura 4.

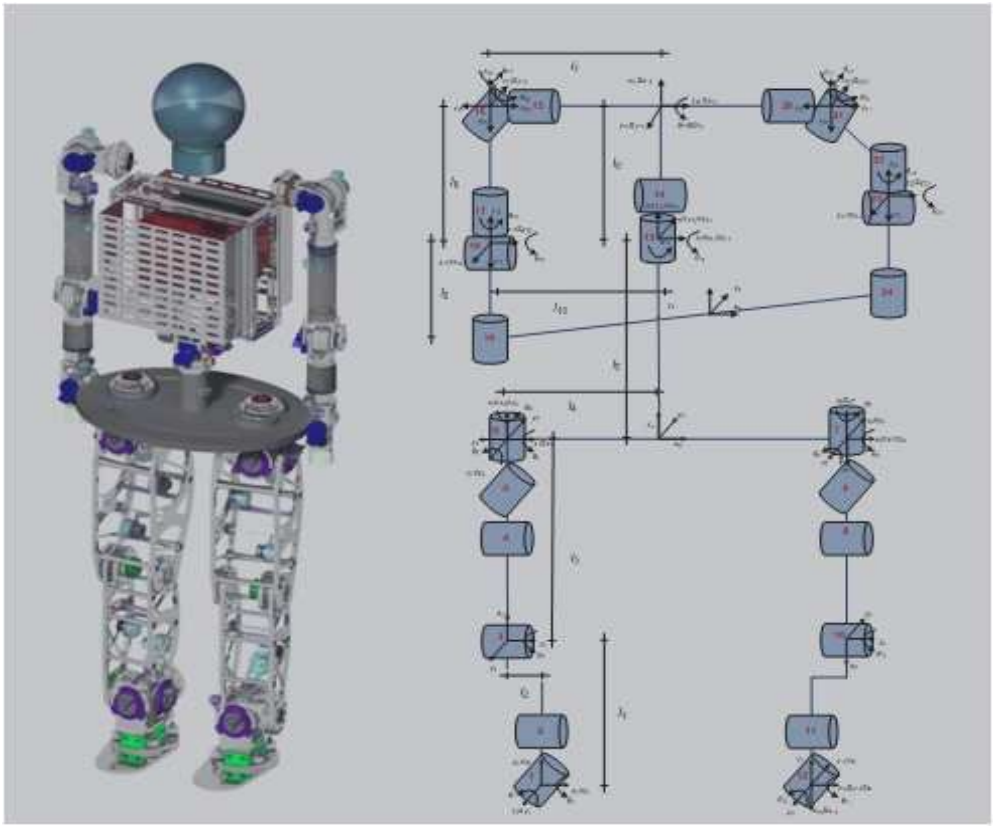


Figura 4: Grados de libertad RH2

Tabla 2: Grados libertad RH2

GDL	Número	Eje de movimiento
PIERNAS	12 GDL	
Cadera	3 (x2)	Sagital Frontal Transversal
Rodilla	1 (x2)	Sagital
Tobillo	2 (x2)	Sagital Frontal
BRAZOS	10 GDL	
Hombros	2 (x2)	Sagital Frontal
Codo	2 (x2)	Sagital Transversal (nuevos)
Muñeca	1 (x2)	Transversal
TRONCO	2 GDL	
Tronco	2	Transversal Sagital (nuevo)
CABEZA	2 GDL	
Cámara	2	Trasversal (izquierda-derecha) Sagital(arriba-abajo)
TOTAL	26 GDL con cabeza 24 GDL sin cabeza	

1.3 Objetivos

Para desarrollar este proyecto hay que realizar una comunicación a través del protocolo CANOpen entre el robot y el PC, por medio del lenguaje de programación C en un sistema operativo Linux. Y realizar una interfaz para el uso de la aplicación por el usuario

De manera más específica, los objetivos principales del proyecto son:

- Implementar un control directo para el robot RH-2 bajo la plataforma del lenguaje C.

- Desarrollar un software que realice el control y la monitorización de los motores que controlan las articulaciones del robot humanoide mediante el protocolo de comunicación CANOpen.

- Desarrollar mediante la herramienta de Matlab una interfaz para la que el usuario pueda interactuar.

- Realizar los ensayos y demostraciones oportunas para la comprobación del correcto funcionamiento del software desarrollado y del hardware empleado.

1.4 Estructura y contenido de la memoria

El proyecto está dividido en siete capítulos:

Capítulo 1. Introducción: como se ha estado diciendo hasta ahora en este capítulo se cuenta la historia del proyecto y los antecedentes de este.

Capítulo 2. Elementos hardware: se realiza una explicación de los elementos hardware como el driver ISCM8005.

Capítulo 3. Elementos software: se realiza una explicación de los elementos software utilizados, como el protocolo de comunicación CANOpen.

Capítulo4. Desarrollo de la aplicación de control directo de ejes: se expone el proyecto realizado y sus diferentes partes así como la interfaz realizada para el usuario.

Capítulo 5. Puesta en marcha de la aplicación: se exponen las gráficas de los resultados obtenidos.

Capítulo6. Conclusiones: se realiza un breve resumen de los objetivos cumplidos y las posibles ampliaciones del proyecto.

Capítulo 7. Presupuesto: se expone el presupuesto realizado para este proyecto.

CAPITULO 2: ELEMENTOS HARDWARE

2.1 DRIVER ISCM8005

2.1.1 Introducción

El driver (actuador) es el encargado de regular el flujo de intensidad que recibe el motor para poder realizar los movimientos del robot de forma deseada.
Para ello se usa el driver ISCM8005, figura 5, debido a sus pequeñas dimensiones para poder introducirlo en el robot y proporcionar la potencia suficiente a los motores.
Esta tarjeta ha sido desarrollada por la empresa Technosoft, posee un lenguaje propio llamado Technosoft Motion Language (TML) con el que se pueden crear sus propios programas que se almacenan en la memoria del actuador.

Se analizarán los modos posibles de movimiento, siempre enfocados para una programación en protocolo CANOpen. La razón de esto es que es el protocolo que se va usar en la red de comunicaciones de CANBus que une la unidad de control con cada uno de los drivers.

Figura 5: Driver ISCM8005

2.1.2 Características

El ISCM8005, [ver referencia 1], es un driver inteligente que proporciona un controlador de movimiento con funciones PLC incluidas y un amplificador, en una unidad muy compacta. Del tamaño de una tarjeta de crédito, el formato de los drives ISCM es ideal para montar en aplicaciones donde se dispone de un espacio muy reducido. El uso de un conector tipo ranura en lugar de contactos atornillados ayuda a desarrollar máquinas en corto tiempo, pudiéndose diseñar tarjetas PCB para alojar todos los componentes y cableado. Gracias a su controlador integrado, los ISCMs pueden fácilmente ser programados para ejecutar de forma autónoma complejos comandos de control de movimiento TML, en modo “stand-alone” o bien en configuraciones de múltiples ejes vía CAN/CANOpen.

El tamaño del driver se pude observar en la figura 6.

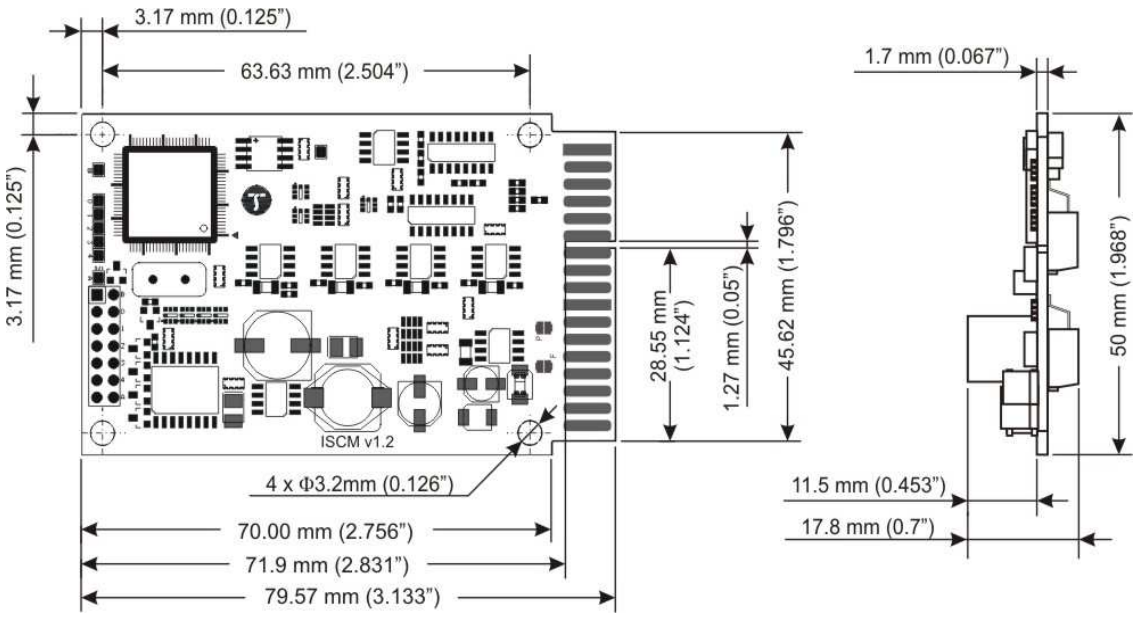


Figura 6: Tamaño del driver

Sus características técnicas destacables:

- Es adecuado para motores CC, brushless y brushed y motores paso a paso.
- Amplificador y controlador de movimiento con funciones PLC en una sola unidad.
- La tensión de alimentación de la lógica de la placa es de 12-48V y de hasta 80V la del motor.
- Capacidad para soportar altas corrientes (5A continuos y 16A de pico) necesarios para ejecutar los pares de movimiento del robot.
- Entradas / Salidas que pueden ayudar a descentralizar el control. Son las siguientes:
    - Interfaz para sensores Hall lineales.
    - Hasta 8 entradas/salidas digitales programables.
    - 2 entradas analógicas con rangos de 0 a5V y +/- 10V.
    - Cuadratura para encoder diferencial e interfaz Hall digital.

- Encoder incremental, sensores Hall y Halls lineales.

- Tiene conexión a puerto serie RS-232 que es necesaria para una inicialización del driver y su correcto funcionamiento con el motor y una conexión a puerto CAN 2.0A y 2.0B hasta 1Mbps. Esta característica es básica puesto que el humanoide tiene la necesidad de un bus de datos con una capacidad de transmisión lo suficientemente elevada como para poder gestionar todos los recursos del humanoide en tiempo real.

- Está diseñado con protecciones específicas contra cortocircuitos, sobre- intensidades, sobre-tensiones y fallos de masa que ayudarán a que no se dañe el driver.

Las características de control del driver son:

- Posee distintos modos de control: torque, velocidad, posición; emulador de motor paso a paso; control de variables externas (presión, flujo, temperatura etc.).

- Permite el uso de un lenguaje propio de Technosoft (TML) que aporta una serie de instrucciones para definición y ejecución de secuencias de movimiento. Este uso permite crear funciones y programas que pueden ser llamados desde la unidad de control. También se pueden predefinir tablas que estén cargadas en la memoria EEPROM del driver y se puedan ejecutar en cualquier momento.

2.1.2.1 Unidades internas del driver

Este dispositivo cuenta con sus propias unidades internas para realizar los cálculos para ello hay que pasar los grados y las revoluciones a las unidades del sistema utilizando las siguientes formulas:

- Para grados se sigue la ecuación 1.

$$Pui = \frac{Pg \times 4 \times N \times Tr}{360} \quad (1)$$

Dónde: Pui: la variable donde se almacena.
Pg: los grados a convertir.
N: número de líneas del encoder (en este caso 500).
Tr: la relación de transformación (en este caso 320).

- Para rpm se sigue la ecuación 2.

$$Vui = \frac{Vrpm \times 4 \times N \times Tr \times T}{60} \quad (2)$$

Dónde: Vui: la variable donde se almacena.
Vrpm: las rpm a convertir.
N: número de líneas del encoder (en este caso 500).

Tr: la relación de transformación (en este caso 320).
T: 0.001(1ms).

2.1.3 Extensión board

Como las conexiones directas del Servo Drive son de difícil acceso y poco manejables, es necesario utilizar, en un primer momento y para realizar las pruebas, una Extension Board, figura 7, para así facilitar el conexionado y ser más manejable de cara a la persona que lo manipule.

Una vez terminado el periodo de pruebas se podría optar por seguir utilizando la Extension Board dentro de la implementación final del prototipo del robot humanoide o, si el espacio lo requiere, utilizar directamente las conexiones de la Intelligent Servo Drive soldando las conexiones.

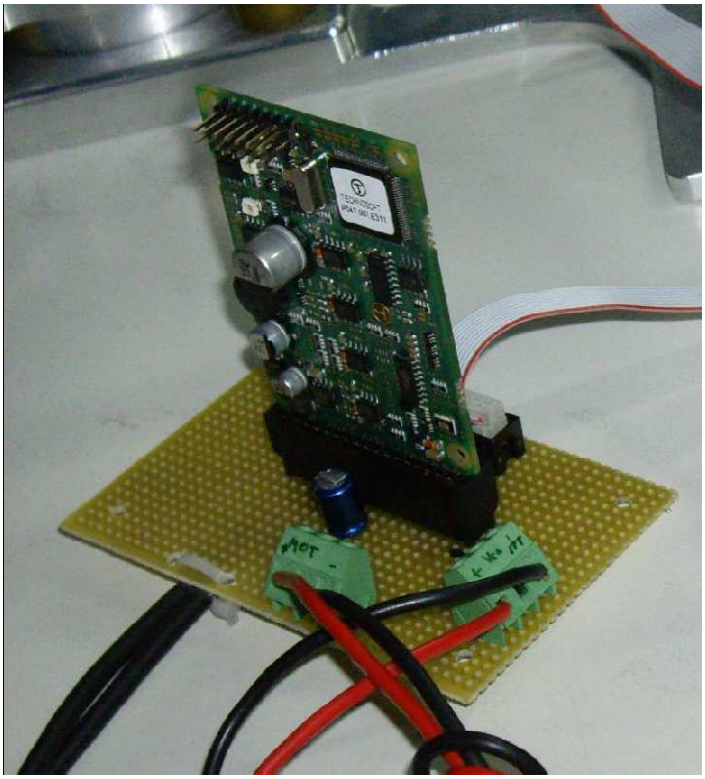


Figura 7: Extensión board desarrollada en al uc3m

2.1.3.1 Conexionado

Estos son los diferentes pines de la tarjeta ISCM8005 y la función que ejercen en ella. En los anexos se puede ver una explicación de los diferentes nombres que están asociados con cada pin

En la cara A, que se muestra en la figura 8, se observan los siguientes pines:

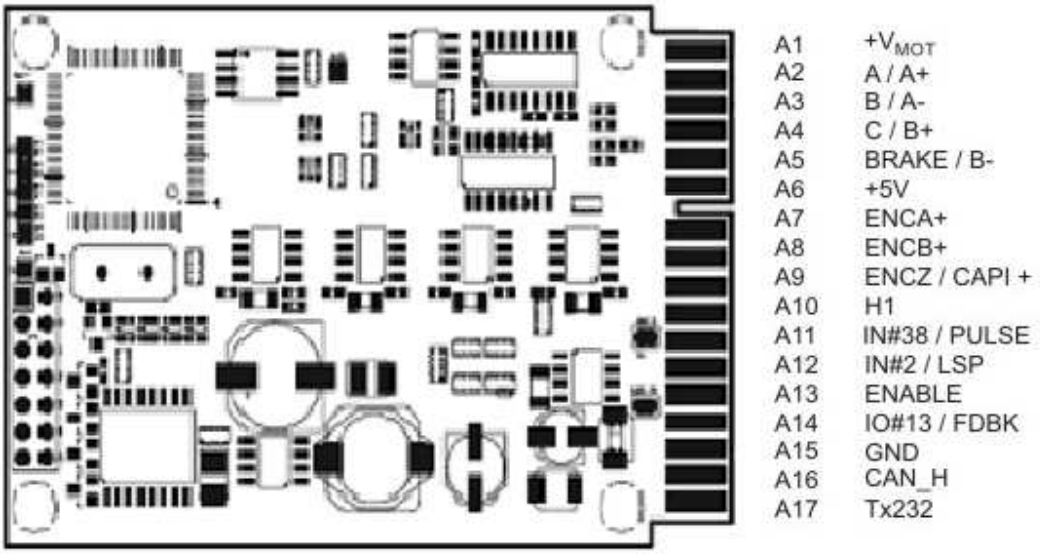


Figura 8: Cara A ISCM8005

Mientras que en la cara B, figura 9, los pines son los siguientes:

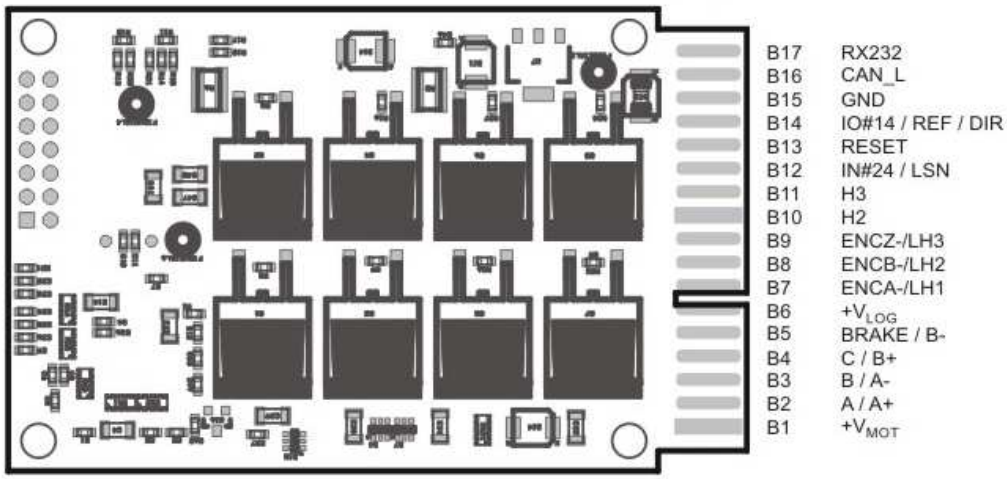


Figura 9: Cara B ISCM8005

2.1.3.1.1 Motores

Al driver se le pueden conectar multitud de tipos de motores: brushless (sin escobillas), paso a paso de 2 y 3 fases y brushed (con escobillas). Sin embargo, para el prototipo RH-2 sólo se van a utilizar motores brushed, figura 10, con la posibilidad de utilizar también brushless, figura 10, por lo que solo se analizarán las conexiones de estos dos tipos de motores.

- Motor brushed: el conexionado se realiza como muestra la figura 9, conectado el pin 2 y pin 3 de la cara A con el positivo y negativo del motor respectivamente, y conectando el pin 15 de las dos caras a tierra.

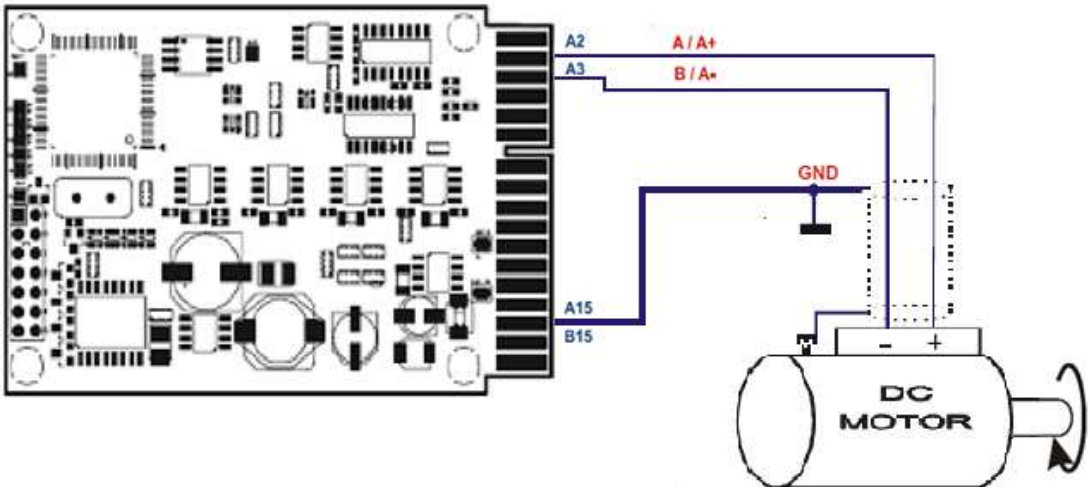


Figura 10: Conexionado motor brushed

- Motor brushless el conexionado se realiza como muestra la figura 11, conectado el pin 2, pin 3 y pin 4 de la cara A con el A, B y C del motor respectivamente, y conectando el pin 15 de las dos caras a tierra.

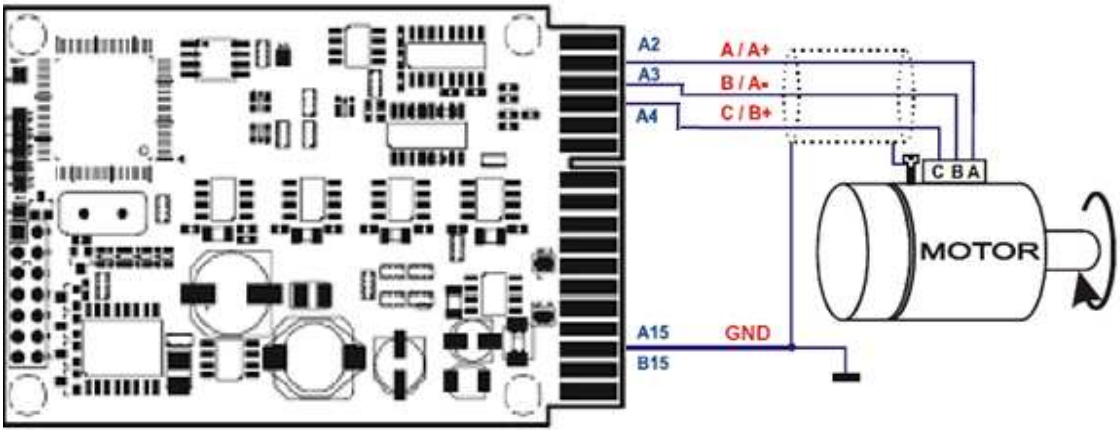


Figura 11: Conexionado motor brushless

2.1.3.1.2 Alimentación

Este dispositivo requiere una fuente de alimentación para la potencia de entre 12 y 80 V y otra para la electrónica digital de entre 12 y 48 V.

El conexionado realizado, figura 12, consiste en conectar ambas tomas a la misma fuente de 24 V para simplificar el circuito final y un condensador a la entrada de la alimentación de valor igual o superior a 100 µF.

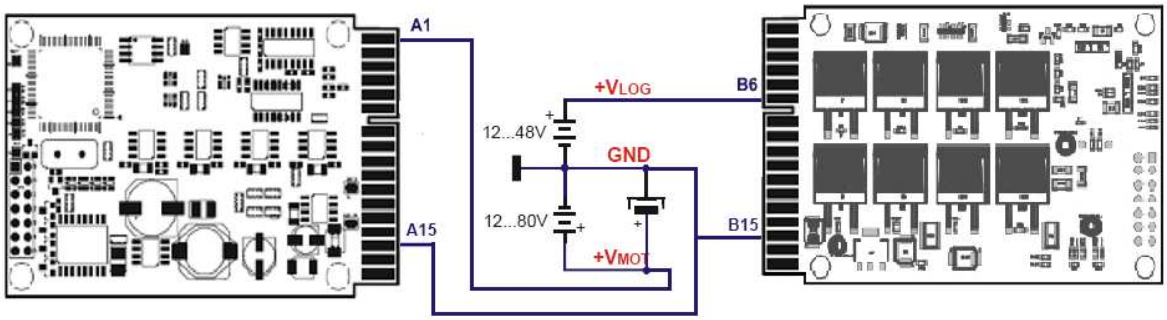


Figura 12: Conexionado alimentación

2.1.3.1.3 RS232

Este conexionado, figura 13, se realiza únicamente para realizar la configuración inicial del driver mediante el programa EasySetUp proporcionado por TechnoSoft. Para conectar el driver al puerto serie de un PC que tenga el software instalado, se ha empleado un conector DB-9

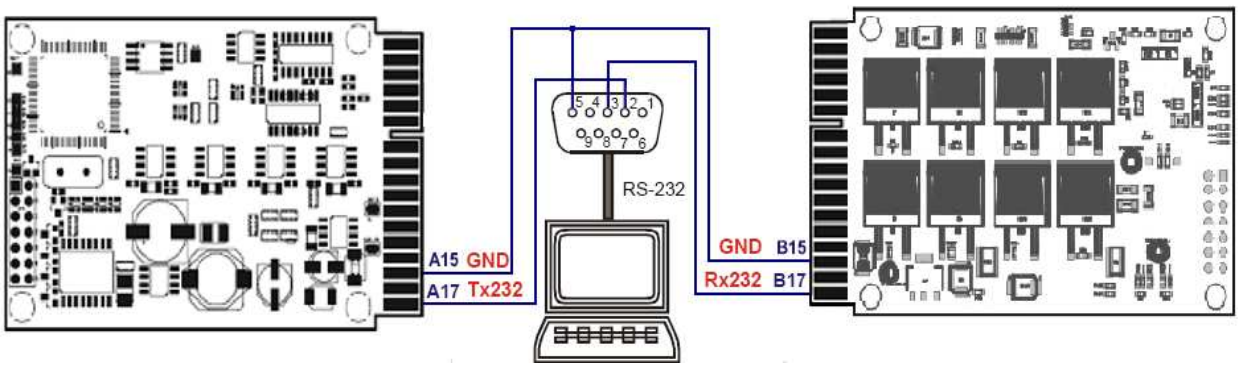


Figura 13: Conexionado RS-232

2.1.3.1.4 CANBus

Para que el driver pueda utilizar el bus de campo CAN es necesario realizar el conexionado de los pines del driver CAN_H, CAN_L y GND a un conector DB-9 de la misma manera que lo implementa la tarjeta CAN, como se muestra en la figura siguiente:

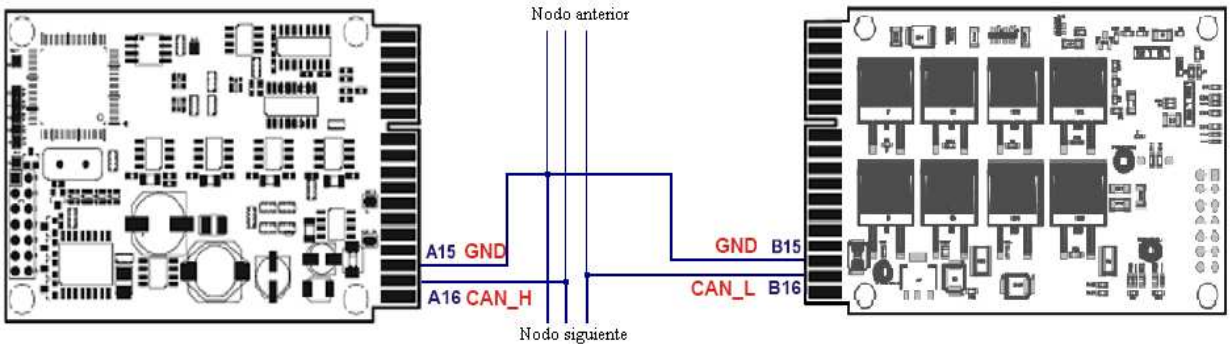


Figura 14: Conexionado CAN

2.1.3.2 Cableado de la red CAN:

Realizar la red CAN mediante dos pares de cables trenzados (2 hilos / par) de la siguiente manera: un par de CAN_H con CAN_L y el otro par de CAN_V+ con CAN_GND, figura 15. La impedancia del cable debe ser de 120 ohmios y una capacidad por debajo de 30pF/metro.

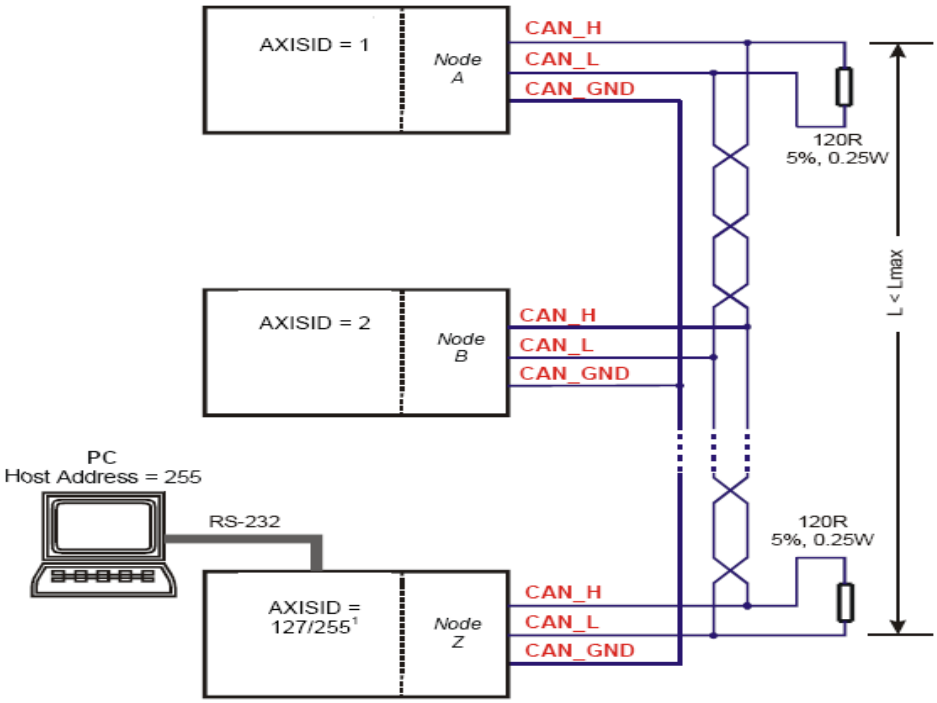


Figura 15: Cableado CAN

2.2 Maqueta de la estructura mecánica del tobillo del RH-2

Como se comentó en la introducción de este documento, dado que el robot humanoide RH-2 se encuentra aún en fase de desarrollo, todos los diseños se han realizado para controlar la estructura mecánica de dos GDL semejante al tobillo del robot humanoide mostrada en la siguiente figura:

Figura 16: Estructura mecánica del tobillo del RH-2

Cada articulación está gobernada por un motor independientemente. Además, para satisfacer las necesidades de potencia y par, cada articulación cuenta con un sistema de reducción formado por un sistema Harmonic Drive con una relación de transmisión de 160 y un sistema correa-poleas con una relación de transmisión de 2, lo que hace un total de una relación de transmisión de 320 entre el motor y la articulación.

2.2.1 Actuación y sensorización

El sistema 315586 de Maxon, figura17, es un sistema que integra en un único bloque un dispositivo actuador (motor brushed 273757) y un dispositivo sensor (encoder relativo 228452). Esto hace que sea un sistema más fiable y robusto, además de facilitar notablemente su integración en el tobillo del RH-2 al prescindir de elementos auxiliares para su interconexión. En la siguiente figura se puede observar este sistema modular, donde se puede diferenciar claramente los dispositivos que lo integran:

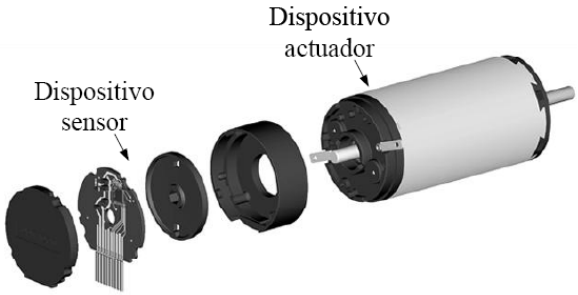


Figura 17: Sistema 315586 de Maxon

2.2.2 Motor brushed

Este motor, figura 18, pertenece al programa RE de motores Maxon DC equipados con potentes imanes permanentes. Tiene un diámetro de 35 mm, escobillas (brushed) de grafito y una potencia de 90W. En la siguiente figura se puede observar el interior de este tipo de motores señalando las partes más importantes que lo componen. En la tabla 3 se muestran alguna de sus características más importantes obtenidas del catálogo facilitado por el fabricante:

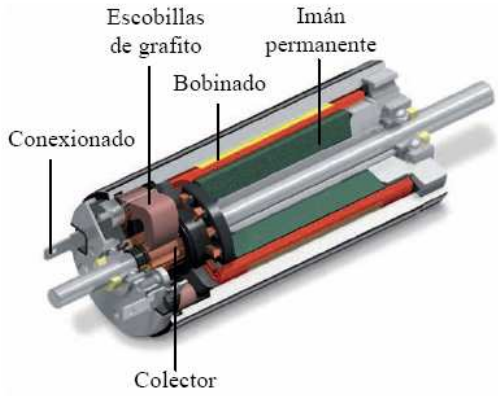


Figura 18: Motor brushed 273757

Tabla 3: Motor brushed 273757

Valores a tensión nominal		Características	
Propiedad	*valor*	*Propiedad*	*valor*
Tensión nominal (V)	48.0	Resistencia en bornes (Ω)	4.75
Velocidad en vacío (rpm)	5960	Inductancia en bornes (mH)	1.29
Corriente en vacío (mA)	59.7	Constante de par (mNm/A)	75.8
Velocidad nominal (rpm)	5150	Constante de velocidad (rpm/V)	126
Par nominal(máx.permanente)(mNm)	98.8	Relación velocidad par (rpm/mNm)	7.89
Corriente nominal(máx.continuo)(A)	1.36	Constante de tiempo mecánica (ms)	5.39
Par de arranque (mNm)	766	Inercia del rotor (gcm^2)	65.2
Corriente de arranque (A)	10.1		
Máx. rendimiento (%)	84		

2.2.3 Motor brushless EC 45 50w flat motor

Este tipo de motor está destinado a ser utilizado como engranajes planetarios ya que tiene un montaje rápido y fácil que no requiere herramientas especiales [ver referencia 8]. En la figura 19 se puede observar el motor y sus características están en la tabla 4.

Figura 19: Motor brushless EC 45 50w flat motor

Tabla 4: Motor brushless EC 45 50w flat motor

PROPIEDADES	VALOR
Voltaje	24Vdc
Potencia	50W
Velocidad de salida	6700rpm
Par de salida	8.43Ncm
Velocidad constante	285 rpm/A

2.2.4 Encoder relativo

Este encoder relativo es de tipo magnético-resistivo (MR) ya que posee un disco magnético multipolar montado en el eje del motor que genera una variación de tensión sinusoidal en el sensor MR.

En la figura 20 se puede observar el interior de este tipo de encoders señalando las partes más importantes que lo componen y en la tabla 5 alguna de sus características más importantes obtenidas del catálogo facilitado por el fabricante:

Figura 20: Encoder relativo 228452

Tabla 5: Encoder relativo 228452

Propiedad	*Valor*
Número de pulsos por vuelta	500
Número de canales	3
Máx. frecuencia de funcionamiento (kHz)	200
Máx. velocidad (rpm)	24000
Tensión de alimentación (V)	5±0.05
Señal de salida	TTL compatible
Desfase (°)	90°e ± 45°e
Anchura de pulso índice (°)	90°e ± 45°e
Rango de temperaturas (°C)	-25 : +85
Momento inercia de rueda de código (gcm2)	1.7
Corriente máx. de salida por canal (mA)	5

CAPITULO 3: ELEMENTOS SOFTWARE

3.1 CANBUS

3.1.1 Introducción

La comunicación entre la CPU y los actuadores del robot debe ser rápida y fiable. Debe controlar el tráfico masivo de información y actuar atendiendo a las preferencias de los mensajes. La conexión tiene que ser sencilla y con poco cableado, ya que los drivers se encuentran en zonas móviles y hay que evitar problemas durante el movimiento.

La comunicación debe ser inmune al ruido, ya que el robot está rodeado de cables y de circuitos electrónicos.

Se ha elegido la comunicación por medio de CANBus, concretamente el protocolo CANOpen. Un protocolo muy utilizado en industrias para sistemas de tiempo real, así como en los sistemas eléctricos del sector automovilístico.

3.1.2 Definición

CANBus es un protocolo de comunicación en serie desarrollado por Bosch para el intercambio de información entre unidades de control electrónicas del automóvil. CAN significa Controller Area Network (Red de área de control) y Bus, en informática, se entiende como un elemento que permite transportar una gran cantidad de información.
Este sistema permite compartir una gran cantidad de información entre las unidades de control abonadas al sistema, lo que provoca una reducción importante tanto del número de sensores utilizados como de la cantidad de cables que componen la instalación eléctrica.

De esta forma aumentan considerablemente las funciones presentes en los sistemas del automóvil donde se emplea el CANBus sin aumentar los costes, además de que estas funciones pueden estar repartidas entre dichas unidades de control.

3.1.3 Características principales de protocolo CAN

- La información que circula entre las unidades de mando a través de los dos cables (bus) son paquetes de 0 y 1 (bit) con una longitud limitada y con una estructura definida de campos que conforman el mensaje.

- Uno de esos campos actúa de identificador del tipo de dato que se transporta, de la unidad de mando que lo trasmite y de la prioridad para trasmitirlo respecto a otros. El

mensaje no va direccionado a ninguna unidad de mando en concreto, cada una de ellas reconocerá mediante este identificador si el mensaje le interesa o no.

- Todas las unidades de mando pueden ser trasmisoras y receptoras, y la cantidad de las mismas abonadas al sistema puede ser variable (dentro de unos límites).

- Si la situación lo exige, una unidad de mando puede solicitar a otra una determinada información mediante uno de los campos del mensaje (trama remota o RDR).

- Cualquier unidad de mando introduce un mensaje en el bus con la condición de que esté libre, si otra lo intenta al mismo tiempo el conflicto se resuelve por la prioridad del mensaje indicado por el identificador del mismo.

- El sistema está dotado de una serie de mecanismos que aseguran que el mensaje es trasmitido y recepcionado correctamente. Cuando un mensaje presenta un error, es anulado y vuelto a trasmitir de forma correcta, de la misma forma una unidad de mando con problemas avisa a las demás mediante el propio mensaje, si la situación es irreversible, dicha unidad de mando queda fuera de servicio pero el sistema sigue funcionando.

- Permite construir sistemas inteligentes. Si un nodo de la red tiene un fallo, el resto de la red sigue operativa.

De acuerdo al modelo de referencia OSI, la arquitectura de protocolos CAN incluye tres capas: física, de enlace de datos y aplicación, además de una capa especial para gestión y control del nodo llamada capa de supervisor.

- Capa física**:** define los aspectos del medio físico para la transmisión de datos entre nodos de una red CAN, los más importantes son niveles de señal, representación, sincronización y tiempos en los que los bits se transfieren al bus.

- Capa de enlace de datos: define las tareas independientes del método de acceso al medio, además debido a que una red CAN brinda soporte para procesamiento en tiempo real a todos los sistemas que la integran, el intercambio de mensajes que demanda dicho procesamiento requiere de un sistema de transmisión a frecuencias altas y retrasos mínimos. En redes multimaestro, la técnica de acceso al medio es muy importante ya que todo nodo activo tiene los derechos para controlar la red y acaparar los recursos. Por lo tanto la capa de enlace de datos define el método de acceso al medio así como los tipos de tramas para el envío de mensajes.

- Capa de aplicación: existen diferentes estándares que definen la capa de aplicación; algunos son muy específicos y están relacionados con sus campos de aplicación. Entre las capas de aplicación más utilizadas cabe mencionar CAL, CANOpen, DeviceNet, SDS (Smart Distributed System), OSEK, CANKingdom.

- Capa de supervisor: la sustitución del cableado convencional por un sistema de bus serie presenta el problema de que un nodo defectuoso puede bloquear el funcionamiento

del sistema completo. Cada nodo activo transmite una señal cuando detecta algún tipo de error y puede ocasionar que un nodo defectuoso pueda acaparar el medio físico. Para eliminar este riesgo el protocolo CAN define un mecanismo autónomo para detectar y desconectar un nodo defectuoso del bus, dicho mecanismo se conoce como aislamiento de fallos.

3.1.4 Componentes del sistema CAN

3.1.4.1 Cables

La información circula por dos cables trenzados que unen todas las unidades de control que forman el sistema. Esta información se trasmite por diferencia de tensión entre los dos cables, de forma que un valor alto de tensión representa un 1 y un valor bajo de tensión representa un 0. La combinación adecuada de unos y ceros conforman el mensaje a trasmitir [ver referencia 9].

En un cable los valores de tensión oscilan entre 0V y 2.25V, por lo que se denomina cable L (Low) y en el otro, el cable H (High) lo hacen entre 2.75V y 5V, como se observa en la figura 21. En caso de que se interrumpa la línea H o que se derive a masa, el sistema trabajará con la señal de Low con respecto a masa, en el caso de que se interrumpa la línea L, ocurrirá lo contrario. Esta situación permite que el sistema siga trabajando con uno de los cables cortado o comunicado a masa. Incluso con ambos comunicados también sería posible el funcionamiento, quedando fuera de servicio solamente cuando ambos cables se cortan.

Es importante tener en cuenta que el trenzado entre ambas líneas sirve para anular los campos magnéticos, por lo que no se debe modificar en ningún caso ni el paso ni la longitud de dichos cables.

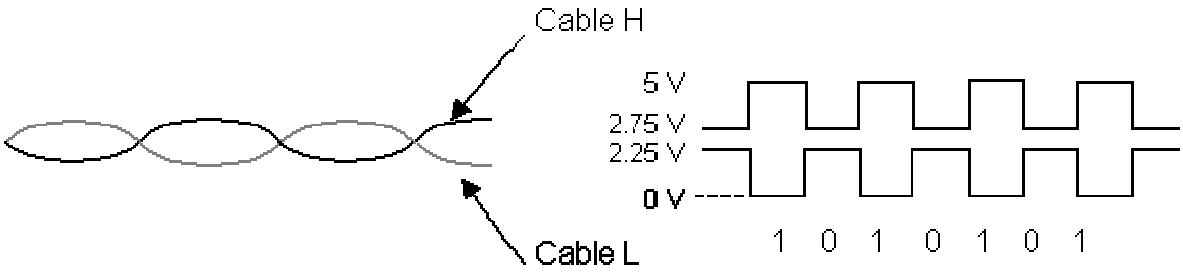


Figura 21: Cables CAN

3.1.4.2 Elemento de cierre o terminador

Son resistencias conectadas a los extremos de los cables H y L, figura 22. Sus valores se obtienen de forma empírica y permiten adecuar el funcionamiento del sistema a diferentes longitudes de cables y número de unidades de control abonadas, ya que impiden fenómenos de reflexión que pueden perturbar el mensaje.

Estas resistencias están alojadas en el interior de algunas de las unidades de control del sistema por cuestiones de economía y seguridad de funcionamiento.

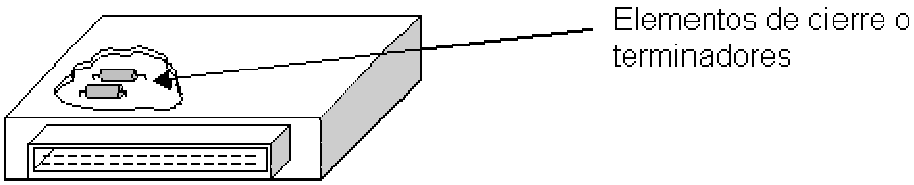


Figura 22: Elemento de cierre o terminador

3.1.4.3 Controlador

Es el elemento encargado de la comunicación entre el microprocesador de la unidad de control y el trasmisor-receptor, figura 23. Trabaja acondicionando la información que entra y sale entre ambos componentes.

El controlador está situado en la unidad de control, por lo que existen tantos como unidades estén conectados al sistema. Este elemento trabaja con niveles de tensión muy bajos y es el que determina la velocidad de trasmisión de los mensajes, que será más o menos elevada según el compromiso del sistema. Así, en la línea de CANBus del motor-frenos-cambio automático es de 500 K baudios, y en los sistema de confort de 62.5 K baudios. Este elemento también interviene en la necesaria sincronización entre las diferentes unidades de mando para la correcta emisión y recepción de los mensajes.

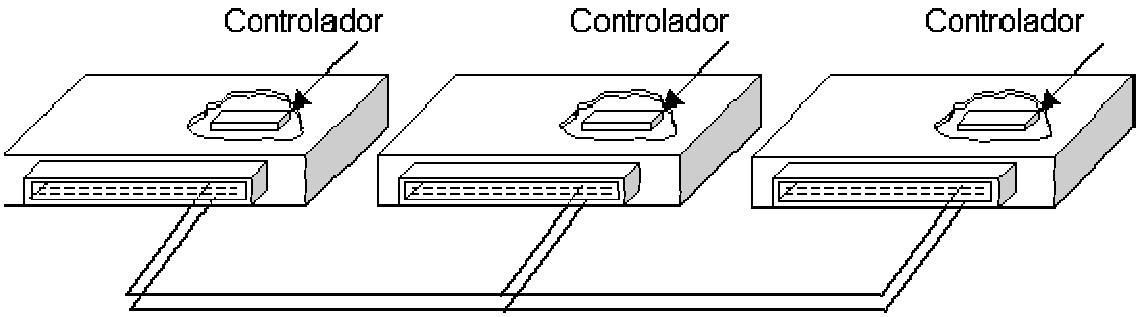


Figura 23: Controlador

3.1.4.4 Transmisor / Receptor

El trasmisor-receptor es el elemento que tiene la misión de recibir y de trasmitir los datos, además de acondicionar y preparar la información para que pueda ser utilizada por los controladores. Esta preparación consiste en situar los niveles de tensión de forma adecuada, amplificando la señal cuando la información se vuelca en la línea y reduciéndola cuando es recogida de la misma y suministrada al controlador.

El trasmisor-receptor, figura 24, es básicamente un circuito integrado que está situado en cada una de las unidades de control abonadas al sistema, trabaja con intensidades próximas a 0.5A y en ningún caso interviene modificando el contenido del mensaje. Funcionalmente está situado entre los cables que forman la línea CANBus y el controlador.

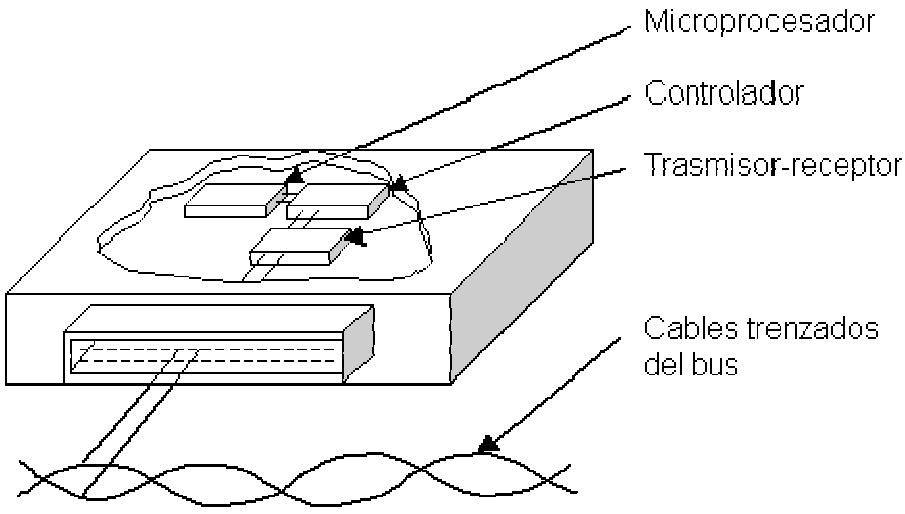


Figura 24: Transmisor / Receptor

3.1.5 Topología del CANBus

El sistema CANBus está orientado hacía el mensaje y no al destinatario. La información es transmitida en forma de mensajes estructurados. Una parte del propio mensaje es un identificador que indica la clase de dato que es. Todas las unidades conectadas al can reciben el mensaje, lo filtran y solo lo emplean las unidades que lo necesitan. Todos los nodos conectados al sistema son capaces de enviar y recibir datos de la línea.

Cuando el bus está libre cualquier unidad conectada puede empezar a transmitir un nuevo mensaje. Si coinciden varias unidades, en la introducción de un mensaje, lo hará la que tenga mayor prioridad. Esta prioridad viene indicada por el identificador. El proceso de transmisión de datos se desarrolla siguiendo un ciclo de varias fases:

1. Suministro de datos: la unidad recibe información de los sensores que tiene asociados (velocidad, posición…) y su microprocesador pasa la información al controlador donde es gestionada y acondicionada para a su vez ser pasada al transmisor-receptor donde se transforma en señales eléctricas.

2. Transmisión de datos: el controlador de dicha unidad transfiere los datos y su identificador junto con la petición de inicio de transmisión, asumiendo la responsabilidad de que el mensaje sea transmitido correctamente a todos los nodos. Para transmitir se ha tenido que encontrar el bus libre y en caso de simultaneidad tener prioridad mayor. Cuando ocurre esto los demás nodos se convierten en receptores.

3. Recepción de datos: las unidades reciben el mensaje y verifican el identificador para comprobar si el mensaje va a ser utilizado por ellos. Las unidades de mando que necesiten los datos del mensaje lo procesan, si no lo necesitan, el mensaje es ignorado.

El sistema CANBus dispone de mecanismos para detectar errores en la trasmisión de mensajes, de forma que todos los receptores realizan un chequeo del mensaje analizando una parte del mismo llamado campo CRC. Otros mecanismos de control que se aplican en las unidades emisoras son la monitorización del nivel del bus, la presencia de campos de formato fijo en el mensaje (verificación de la trama), análisis estadísticos por parte de las unidades de mando de sus propios fallos, etc.

Estas medidas hacen que las probabilidades de error en la emisión y recepción de mensajes sean muy bajas, por lo que es un sistema extraordinariamente seguro. Según las especificaciones del protocolo de Bosch, la probabilidad de error residual de mensajes erróneos no detectado ha de ser inferior a 4.7 x 1011.

3.1.5.1 Formato de tramas CAN.

La información que circula entre las unidades de mando a través del puerto de comunicaciones serie son paquetes de bits con una longitud limitada y con una estructura definida de campos que conforman el mensaje. Uno de los campos actúa de identificador del tipo de dato que se transporta, del nodo que lo trasmite y de la prioridad para trasmitirlo respecto a otros. El mensaje no va direccionado a ningún nodo en concreto, sino que cada uno de los nodos reconocerá mediante este identificador si el mensaje le interesa o no.

La estructura del menaje permite llevar a cabo el proceso de comunicación entre las unidades de mando según el protocolo definido por Bosch para el CANBus, donde los distintos campos que lo compone facilitan desde identificar a la unidad de mando, como indicar el principio y el final del mensaje, mostrar los datos, permitir distintos controles, etc.

Los mensajes son introducidos en la línea con una cadencia que oscila entre los 0.055 y 12.9 milisegundos dependiendo de la velocidad del Bus y de la unidad de mando que los introduce [ver referencia 2].

Estructura del mensaje estándar:

En la figura 25 se puede observar los diferentes campos que forman el mensaje:

- Campo de inicio del mensaje (SOF): el mensaje se inicia con un bit dominante “0”, cuyo flanco descendente es utilizado por las unidades de mando para sincronizarse entre sí.

- Campo de arbitraje: los 12 bits de este campo se emplean como identificador y permite reconocer al nodo que emite la trama, el tipo de mensaje y la prioridad de este. Cuanto más bajo sea el valor del identificador más alta es la prioridad, y por lo tanto determina el orden en el que van a ser introducidos los mensajes en el Bus. El bit RTR indica si el mensaje contiene datos (RTR=0) o si se trata de una trama remota sin datos (RTR=1). Una trama de datos siempre tiene prioridad frente a las tramas remotas que se emplea para solicitar datos a otros nodos, bien porque se necesitan o para realizar un chequeo.

- Campo de control: este campo informa sobre las características del campo de datos. El bit IDE indica cuando es un “0” que se trata de una trama estándar y cuando es un “1” que es una trama extendida. La diferencia entre una trama estándar y una trama extendida es que el identificador de la primera tiene 11 bits y la segunda 29 bits. Ambas tramas pueden coexistir eventualmente, siempre y cuando todos los controladores del sistema soporten el formato extendido, y la razón de su presencia es la existencia de dos versiones de CAN.

- Campo DLC: los cuatro bit que lo componen indican el número de bytes contenido en el campo de datos.

- Campo de datos: en este campo aparece la información del mensaje con los datos que la unidad de mando correspondiente introduce en la línea CANBus. Puede contener entre 0 y 8 bytes (de 0 a 64 bit).

- Campo de chequeo (CRC): este campo tiene un formato predefinido con una longitud de 16 bits de los cuales los 15 primeros son utilizados para la detección de errores, mientras que el último siempre es un bit recesivo “1” que delimita el campo CRC.

- Campo de confirmación (ACK): el campo ACK está compuesto por dos bit que son siempre trasmitidos como recesivos “1”. Todos los nodos que reciben el mismo CRC modifican el primer bit del campo ACK por uno dominante “0”, de forma que la unidad de mando que está todavía trasmitiendo reconoce que al menos uno de los nodos conectados al sistema ha recibido el mensaje correctamente. De no ser así, el nodo transmisor interpreta que su mensaje presenta un error.

- Campo de final de mensaje (EOF): este campo indica el final del mensaje con una cadena de 7 bits recesivos.

- Interframe space (IFS): está compuesto de al menos tres bits recesivos llamados "intermission" y su función es la de separar dos tramas consecutivas. Esto permite el procesado interno de los mensajes antes de recibir una nueva trama. Después del IFS el Bus permanece en un estado recesivo indicando que está libre para una nueva transmisión.

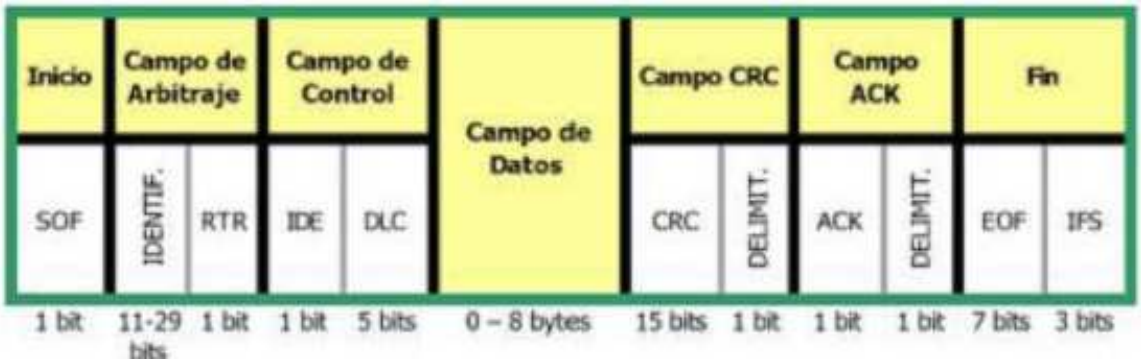


Figura 25: Mensaje CAN

Puede ocurrir que en determinados mensajes se produzcan largas cadenas de ceros o unos, y que esto provoque una pérdida de sincronización entre los distintos nodos. El protocolo CAN resuelve esta situación insertando un bit de diferente polaridad cada cinco bits iguales: cada cinco "0" se inserta un "1" y viceversa. El nodo que utiliza el mensaje, descarta un bit posterior a cinco bits iguales. Estos bits reciben el nombre de bit stuffing.

3.1.5.2 Filtrado de mensajes CAN

El CANBus está basado en un concepto de comunicación Broadcast, lo que significa que cualquier nodo perteneciente a la red pueden escuchar todos los mensajes que se transmiten por ella. Después de recibir un mensaje cada nodo ha de decidir si este es aceptado o no, y por eso es necesario implementar filtros de aceptación en cada uno de los nodos. En la figura 26 se puede observar como el nodo 2 transmite un mensaje y los demás nodos lo reciben. El nodo 3 desecha el mensaje debido a que el filtro no coincide con el identificador de la trama recibida, mientras que los nodos 1 y 4 si lo aceptan, por lo que han de procesar el contenido. También se puede observar que el nodo que transmite la trama monitoriza el Bus para chequeo de errores.

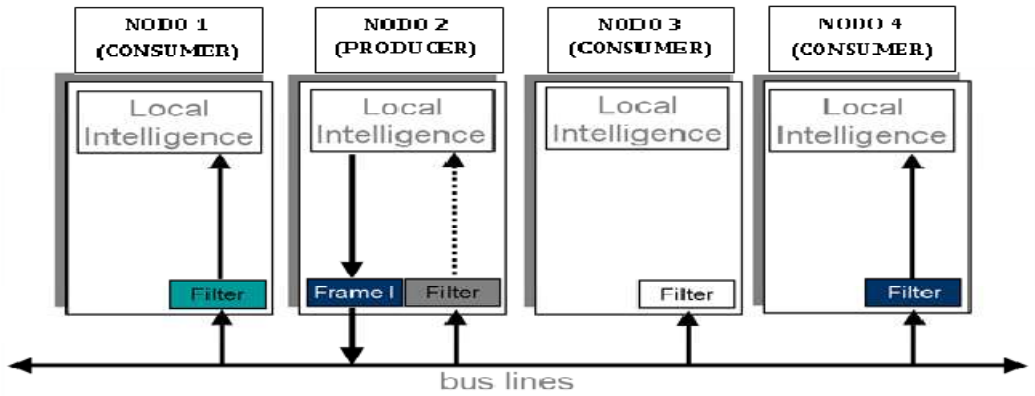


Figura 26: Filtrado de mensajes CAN

3.1.5.3 Diagnosticar el CANBus

Los sistemas de seguridad que incorpora el CANBus permiten que las probabilidades de fallo en el proceso de comunicación sean muy bajas, pero sigue siendo posible que cables, contactos y los propios dispositivos presenten alguna disfunción. Para el análisis de una avería, se debe tener presente que un nodo averiado que este abonado al Bus, en muchos casos no impide que el sistema siga trabajando con normalidad. Lógicamente no será posible llevar a cabo las funciones que implican el uso de información que proporciona la unidad averiada, pero sí todas las demás.

Es posible localizar fallos en el CANBus utilizando sistemas de auto diagnosis, donde se podrá averiguar desde el estado de funcionamiento del sistema hasta los nodos asociadas al mismo, pero necesariamente se ha de disponer del equipo de chequeo apropiado. Otra alternativa es emplear programas informáticos como el CANking, MiniMon, CANAlyzer, etc.

3.2 CANOPEN

3.2.1 Introducción.

El bus de campo CAN solo define la capa física y de enlace del modelo ISO/OSI, por lo que es necesario definir como se asignan y utilizan los identificadores y datos de los mensajes, figura 27. Para ello se definió el protocolo CANOpen, que está basado en CAN e implementa la capa de aplicación [ver referencia 10].

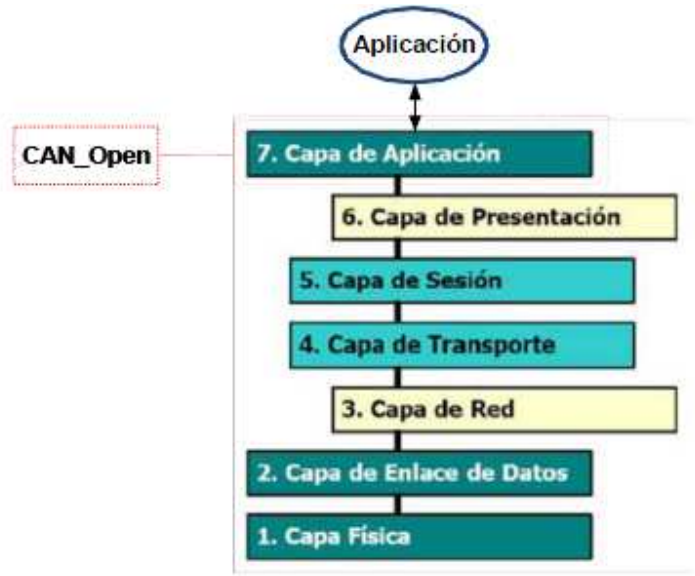


Figura 27: Capas mensaje CANOpen

La capa de aplicación está comprometida con un concepto de configuración, transmisión de datos en tiempo real y el mecanismo de sincronización entre componentes. La funcionalidad que ofrece a la aplicación está dividida en diferentes objetos de comunicación mediante los cuales las aplicaciones interactúan entre sí. Para ello, el objeto de comunicación intercambia datos por el Bus con uno o varios objetos siguiendo un protocolo específico para cada objeto de comunicación.

3.2.2 Modelo del nodo CANOpen

La estructura de un nodo CANOpen, como se puede apreciar en la figura 28, se divide en tres partes:

- Comunicación: la función de esta unidad es suministrar los objetos de comunicación y la apropiada funcionalidad para transportar los campos de datos por el Bus.

- Diccionario de objetos: es una colección de todos los campos de datos que influyen en el desarrollo de las aplicaciones de objetos, los objetos de comunicación y la máquina de estados usada en el nodo.

- Aplicación**:** la aplicación controla la funcionalidad del nodo con respecto a la interacción con el entorno de procesos. El diccionario de objetos cumple con la función de interfaz entre la comunicación y la aplicación. La descripción completa de las aplicaciones de un dispositivo con respecto a las entradas en el diccionario del objeto se conoce como perfil del dispositivo.

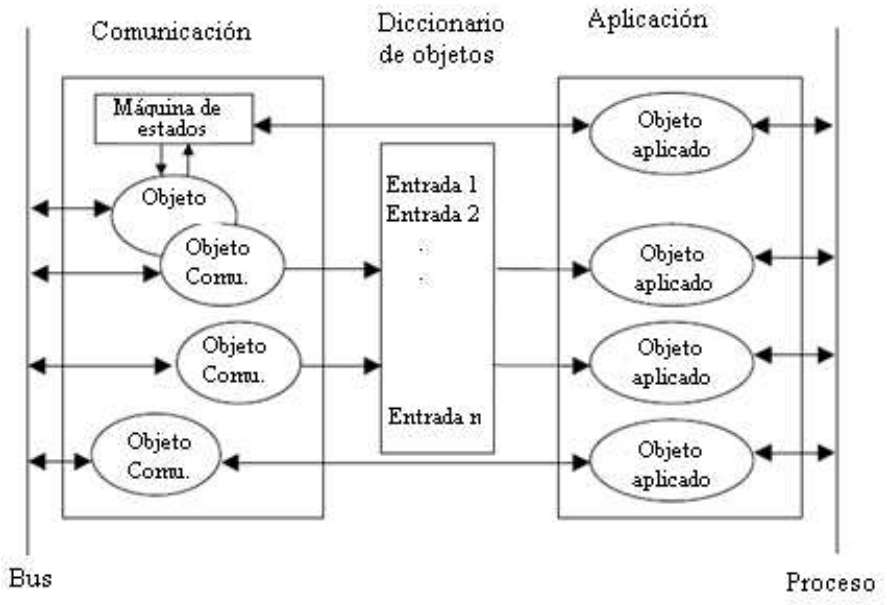


Figura 28: Modelo del nodo CANOpen

El diccionario de objetos cumple con la función de interfaz entre la comunicación y la aplicación. La descripción completa de las aplicaciones de un dispositivo con respecto a las entradas en el diccionario del objeto se conoce como perfil del dispositivo. El diccionario de objetos es una de las partes más importantes por lo que se tratará en profundidad más adelante.

3.2.3 Modelo de comunicación

El modelo de comunicación define los diferentes objetos de comunicación, servicios y modos de transmisión de los mensajes. Este modelo soporta tanto transmisiones síncronas como asíncronas.

Por medio de la transmisión síncrona, la adquisición de datos en la red puede ser completamente coordinada. Este tipo de comunicación esta predefinida para los objetos de comunicación (Sync message, timestamp message). Los mensajes síncronos son transmitidos con respecto a un mensaje de sincronización predefinido, mientras que los mensajes asíncronos pueden ser transmitidos en cualquier momento. Debido al carácter eventual del mecanismo de comunicación es posible definir tiempos de inhibición para la comunicación. Estos tiempos consisten en establecer el tiempo mínimo que tiene que transcurrir entre dos servicios consecutivos del mismo objeto de datos, para garantizar que lo mensajes de baja prioridad acedan a la red durante el tiempo de inhibición. Los tiempos de inhibición pueden ser asignados por la aplicación.

Con respecto a la funcionalidad, se pueden distinguir tres tipos de comunicación:

- Comunicación Master/Slave.
- Comunicación Client/Server.
- Comunicación Producer/Consumer.

3.2.3.1 Comunicación Master/Slave

En este tipo de comunicación solo puede haber un maestro en la red para una funcionalidad específica y en cualquier tiempo dado, figura 29. Todos los demás nodos del Bus están considerados como esclavos. El maestro emite una petición y los esclavos responden a esta petición si el protocolo lo requiere.

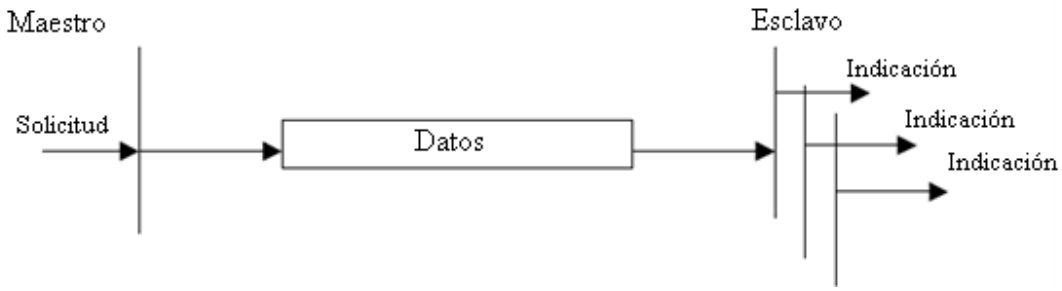


Figura 29: Comunicación Master/Slave.

3.2.3.2 Comunicación Client/Server

Este tipo de comunicación se produce entre dos nodos donde uno de ellos cumple con la funcionalidad de cliente y el otro la de servidor, figura 30. El cliente emite una petición de carga o descarga de datos pidiéndole al servidor que realice una cierta tarea. Después de finalizar la tarea el servidor emite una respuesta a la petición.

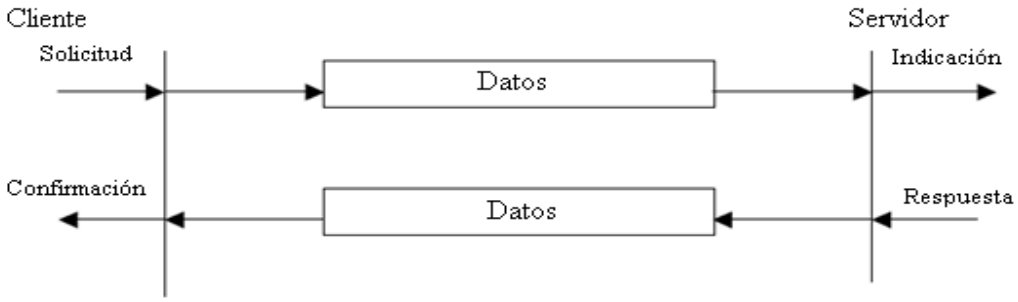


Figura 30: Comunicación Client/Server

3.2.3.3 Comunicación Producer/Consumer

En este tipo de comunicación siempre se ve involucrado un nodo como productor del mensaje y uno o más consumidores de dicho mensaje. Se pueden distinguir dos variantes de este tipo de comunicación:

- Push model, figura 31, en un servicio que no espera respuesta. Se envía el mensaje sin pedir primero una petición del mismo.

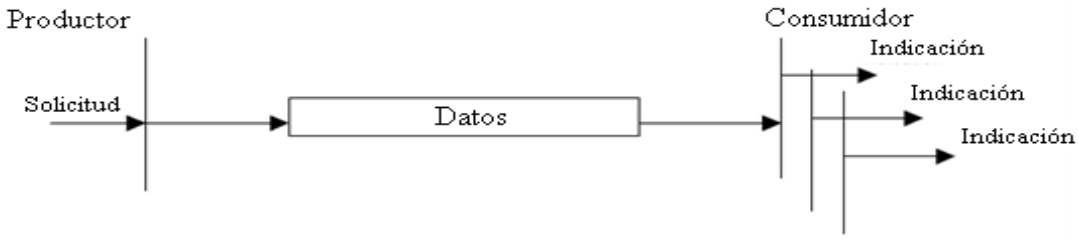


Figura 31: Push model

- Pull model, figura 32, es un servicio que espera una respuesta. Para poder enviar los datos primero tiene que haber una petición para el envío.

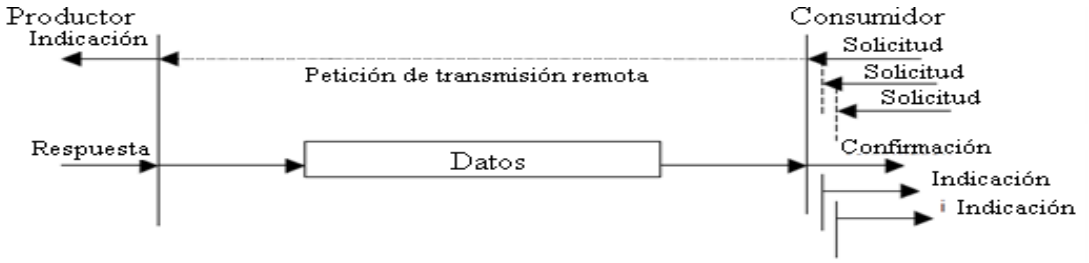


Figura 32: Pull model

3.2.4 Objetos de comunicación

Los objetos de comunicación están descritos por los servicios y protocolos. Los tipos de servicios (confirmado, sin confirmar, etc.) están descritos de forma que contienen los parámetros de servicio primitivo que se definen para cada servicio en particular.

Todos los servicios asumen que no se producirá ningún error, tanto en el envío de datos como en la capa física del bus. Y en el caso de que se produzca algún error, este será resuelto por la aplicación.

Todo el tráfico de datos por el bus relacionado con un dispositivo CANOpen esta conceptualmente dividido en dos clases, el SDO y el PDO. El SDO incluye todo lo relacionado con parámetros de configuración, mientras que el PDO abarca todo lo relacionado con las operaciones de tiempo real del dispositivo.

El protocolo CANOpen requiere identificadores de mensajes tal que todos los PDOs reciban mayor prioridad que los SDOs. A continuación en la tabla 6 se puede observar una comparativa entre ambos tipos de mensajes [ver referencia 3].

Tabla 6: Diferencias entre PDO y SDO

PDO (Process Data Object)	SDO (Service Data Object)
Usado para intercambio de datos en Tiempo Real.	Usado para mensajes de configuración.
Posibilidad de transmisión ciclica o a ciclica.	Transmisión asincrona normalmente.
Optimizado para alta velocidad.	Baja velocidad.
Mensaje de prioridad alta.	Mensaje de prioridad baja.
Solo 8 bytes por mensaje.	Posibilidad de Multi-telegramas.
Campos de datos	Campos de datos
Estructura configurable mediante el canal de servicios.	Estructura fija.
Rápido.	Lento.

Adicionalmente a los SDOs y PDOs existen otros tipos de objetos:

- Sincronización (SYNC): se utiliza para sincronizar todas las aplicaciones del bus.
- Network Management Object (NMT): se encarga de controlar y enviar información a la máquina de estados de los diferentes nodos conectados al bus.
- Time Stamp Object (TIME): provee una referencia de tiempo común a todos los nodos del bus.
- Emergencia (EMCY): envía información de errores internos en los nodos si estos se producen y el objeto esta implementado.

3.2.4.1 Mensajes SDO

El Service Data Object (SDO) permite comunicar los objetos de datos entre el maestro y los demás nodos por medio del acceso al diccionario de objetos del dispositivo CANOpen.

Los SDOs son usados normalmente para configurar el driver después de ser encendido, para mapear PDOs y para comunicación poco frecuente de baja prioridad. El protocolo de transmisión SDO permite transmitir objetos de cualquier tamaño, por lo que puede ser necesario el envío y recepción de varios segmentos, siendo necesaria la confirmación de recepción. El primer byte del primer segmento contiene la información del control del flujo de datos, los siguientes tres bytes el índice y el subíndice del objeto dentro del diccionario al que se desea acceder y los otros cuatro bytes están disponibles para datos del usuario, todo ello precedido del COB-ID pertinente formado por 11 bits. El segundo y los siguientes segmentos enviados, utilizando el mismo COB-ID, contienen un byte de control y siete bytes de datos del usuario. El receptor de los mensajes confirma cada segmento o bloque de segmentos recibido, por lo que es un sistema de comunicación punto a punto.

Los SDO tienen el siguiente formato:

COB-ID	Tamaño de datos (1 byte)	Índice (2 bytes)	Subíndice (1 byte)	Datos (4 bytes)

- Los 4 bits más significativos del COB-ID corresponden al tipo de objeto de comunicación y los 7 menos significativos al nodo del driver
- El COB-ID del SDO transmisor tiene el valor 600 mientras que el SDO receptor tiene el valor 580

Por ello, el COB-ID de los paquetes de SDO recibidos por el driver se calcula como 600h + el ID del nodo. El COB-ID de los paquetes de SDO enviados por el driver es calculado como 580h + el ID del nodo.

3.2.4.1.1 Componer un mensaje SDO

Quieres enviar por un SDO la velocidad a la que tiene que ir el motor, se usa el nodo 1.
Se tiene que acceder al objeto 60FFh "target velocity" subíndice 00 y le quieres enviar la velocidad de 600 rpm, calculada antes.

COB-ID	Tamaño de datos	Índice	Subíndice	Parámetros
601	23	FF 60	00	00 0F 00 00

- COB-ID: 600+1 (nodo elegido).

- Tamaño de datos: 23 (significa que el objeto necesita 32 bits, se explica más adelante.

- Índice: es el objeto elegido (del diccionario de objetos), se coloca del byte menos significativo al más significativo.

- Subíndice: lo indica el objeto, en este caso 0.

- Parámetros: son los datos que se quieren introducir, en este caso 600 rpm que se pasan a unidades internas (esto se explica en el apartado 3) y se transforma el número a hexadecimal. También se colocan del byte menos significativo al más significativo.

3.2.4.2 Mensajes PDO

Son usados para la transferencia de datos puntuales de alta prioridad, entre maestros y esclavos en tiempo real. Aunque estos no están indexados se corresponden con entradas en el diccionario de objetos y provee de una interfaz a los objetos de las aplicaciones. El tipo de dato y mapeado de un PDO para cada objeto de aplicación está determinada por defecto dentro de la estructura mapeada en el diccionario de objeto de cada dispositivo. Si el nodo soporta el mapeado de PDO variable, el formato y contenido de los mensajes PDO pueden ser configurados entre el servidor y cliente en la fase de inicialización del bus. Esta configuración se realiza mediante servicios SDO correspondientes a cada entrada en el diccionario de objetos que se desee modificar. El número y tamaño de los PDO de un dispositivo es específico década aplicación y está definido dentro del perfil específico de cada tipo de dispositivo.

Existen dos tipos de PDO. El primero para transmitir (Transmit PDO) y el segundo para recibir (Receive PDO). Los PDOs están descritos por los parámetros de comunicación PDO (en el index 20h) y por los parámetros de mapeado PDO (en el index 21h). Los parámetros de comunicación describen la capacidad de comunicación de los PDOs (COB-ID utilizado por el PDO, tiempos de inhibición y temporización) y los parámetros de mapeado contienen información acerca del contenido de los PDOs.

Existen 4 TPDOs y 4 RPDOs, el índice de la correspondiente entrada en el diccionario de objetos se calcula aplicando las siguientes formulas:

- RPDO índice de parámetros de comunicación= 1400h + RPDOnúmero-1.
- TPDO índice de parámetros de comunicación= 1800h + TPDOnúmero-1.
- RPDO mapeo del índice de parámetros= 1600h + RPDOnúmero -1.
- TPDO mapeo del índice de parámetros= 1A00h + TPDOnúmero -1.

Los parámetros de comunicación y mapeado son obligatorios para todos los PDOs.

3.2.4.2.1 Mapeo de un PDO

Para mapear un PDO hay que seguir estos pasos:

1. Se deshabilita el PDO. En el Objeto de Mapeo de PDO (índices 1600h-1603h para RPDOs y 1A00h-1A03h para TPDOs) se pone el primer subíndice a 0.
2. Si fuera necesario, se cambian los parámetros de comunicación del PDO (índices 1400h-1403h para RPDOs y 1800h-1803h para TPDOs): el COB-ID, el tipo de transmisión o el acontecimiento que provoca la transmisión.

3. Se mapean los nuevos objetos. Se escribe en el Objeto de Mapeo de PDO'S (de índices 1600h-1603h para RPDOs y 1A00h-1A03h para TPDOs) en los subíndices 1 a 8 la descripción de los objetos que serán apeados. Se pueden mapear hasta 8 objetos de 1 byte de tamaño.

4. Se habilita el PDO. En el subíndice 0 del Objeto de Mapeo asociado al PDO (índices 1600h-1603h para RPDOs y 1A00h-1A03h para TPDOs) se escriben el número de objetos mapeados.

Ejemplo de mapeo de un PDO:

Se quiere mapear el PDO3 receptor de un nodo con ID 1 con Palabra de Control (índice 6040h) y Modo de Operación (índice 6060h).

1. Deshabilitado del RPDO: se escribe 0 en el objeto 1602h, subíndice 0
   Cob-id= 601h (600 + ID del nodo).

Tamaño de datos	Índice	Subíndice	Parámetros
2F	02 16	00	00 00 00 00

2. No hace falta cambiar los parámetros de comunicación.

3. Se mapean los nuevos objetos:

   a- Objeto 1602h, subíndice 01h se escribe la descripción de Control Word.

Tamaño de datos	Índice	Subíndice	Parámetros
23	02 16	01	10 00 40 60

   b- Objeto 1602h, subíndice 02h se escribe la descripción de Modo de Operación.

Tamaño de datos	Índice	Subíndice	Parámetros
23	02 16	02	08 00 60 60

4. Se habilita el PDO: Se escribe 2 en el objeto 1602h, subíndice 00h.

Tamaño de datos	Índice	Subíndice	Parámetros
23	02 16	00	02 00 00 00

3.2.4.3 Mensajes de sincronización (SYNC)

El mensaje de sincronización permite la sincronización de los dispositivos en la red y consigue que la transmisión de PDOs sea síncrona. La periodicidad de estos mensajes se define mediante una herramienta de configuración durante el proceso de reinicio. Estos mensajes tienen un identificador perteneciente al grupo de muy alta prioridad, el 128, y no lleva ningún dato. Está disponible en el índice 1005h del diccionario de objetos.

Durante la ejecución de aplicaciones críticas, que requieren una sincronización más exacta, los drivers pueden usar el protocolo de sincronización opcional de alta resolución, que emplea un formato especial de mensaje. El índice de objeto 1013h contiene sellos de tiempo con resolución de 1µs. Este objeto puede ser mapeado en un PDO para definir un mensaje alta resolución. El PDO debería ser configurado para la transmisión síncrona. Cuando uno de los drivers es puesto como maestro de sincronización, el sello de tiempo de Alta resolución por defecto es enviado usando el COB-ID definido en el objeto COB-IDs de los sellos de tiempo de alta resolución de índice 2004h.

3.2.4.4 Mensajes de emergencia (EMCY)

Un driver envía un mensaje de la emergencia (EMCY) cuando ocurre un error interno. El mensaje de emergencia es transmitido sólo una vez por cada error. Mientras no ocurra ningún nuevo error, el driver no mandará más mensajes de emergencia. Puede que ningún dispositivo sea consumidor de este mensaje, pero pueden ser varios los receptores. El COB-ID con el que se envían los mensajes de emergencia se define con el objeto 1014h. La acción a tomar por el receptor de un mensaje de emergencia es específica de cada aplicación. CANOpen define varios códigos de error de emergencia para ser transmitidos en un mensaje de emergencia, que tiene sólo ocho bytes de datos con la siguiente información.

Código de error de emergencia	Registro de errores (objetos 1001h)	Campo de error específico del fabricante
0-1	2	3-7

Los detalles en cuanto a las condiciones que puedan generar mensajes de emergencia son presentados en el objeto cuyo índice es 2000h “registro de erros de movimiento”.

3.2.4.5 Mensajes time stamp object (TIME)

Este es un objeto que es usado cuando el sistema requiere una alta precisión de sincronismo para ajustar la inevitable deriva de los relojes locales. El identificador de dicho objeto, cuyo valor por defecto es el 256 (100h), está almacenado en el índice 1012h del Diccionario de Objetos.

3.2.4.6 Mensajes Network Management Object (MNT)

Un mensaje NMT es muy sencillo, pues sólo contiene dos bytes de datos y su COB-ID es 0. El primer byte de datos contiene el comando específico de gestión de la red y el segundo el identificador del nodo al que va dirigido. Si este segundo byte tiene valor 0, el mensaje va dirigido a todos los nodos de red. Los comandos NMT se utilizan para controlar el estado de la comunicación del driver.

3.2.5 Estados del driver

Todo dispositivo CANOpen implementa una máquina de estados que describe los estados posibles en los que se puede encontrar y las secuencias de control posibles para pasar de un estado a otro, tal y como se muestra en la figura 33.

Estados:

- Inicialización**:** este estado es donde el nodo entra después de activar el dispositivo ó resetearlo por hardware y es donde los valores de áreas del perfil de fabricación, los valores de área de dispositivo estandarizado y los parámetros del área de comunicaciones recuperan sus valores predefinidos. Tras la inicialización se envía el mensaje de Bootup para pasar al estado siguiente.

- Pre-operacional**:** la configuración de PDOs, los parámetros del dispositivo y la asignación de objetos de aplicación (PDO-maping) pueden ser realizados por medio de SDOs para una determinada configuración de aplicación. El dispositivo puede pasar directamente al estado operacional por medio del servicio arrancar por nodo remoto (Start_Remote_Node) si no necesita ser configurado previamente. En este estado la comunicación por medio de PDOs no está permitida.

- Operacional: en este estado todos los objetos de comunicación están activados y es el estado normal de funcionamiento de un nodo CANOpen.

- Parada: cuando el dispositivo entra en este estado se detiene toda la comunicación a excepción del node guarding o heartbeat, si están activos, y los servicios de control de módulos del NMT.

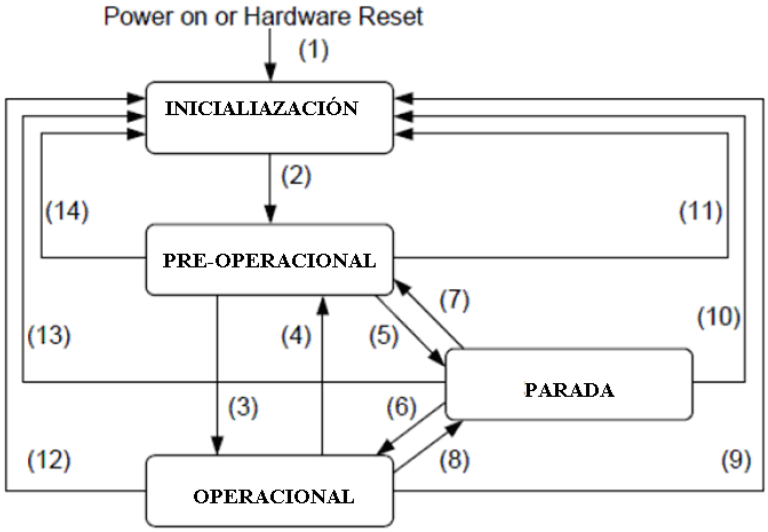


Figura 33: Estados del driver

En la tabla 7 se explican los pasos realizados en la figura 33.

Tabla 7: Estados del driver

(1)	Tras la activar el dispositivo entra en el estado INITIALISATION automáticamente.
(2)	Inicialización finalizada - entra en PRE-OPERATIONAL automáticamente.
(3),(6)	Servicio Start_Remote_Node.
(4),(7)	Servicio Enter_PRE-OPERATIONAL.
(5),(8)	Servicio Stop_Remote_Node.
(9),(10),(11)	Servicio Reset_Node.
(12),(13),(14)	Servicio Reset_Communication.

3.2.6 Construcción de un mensaje

Independientemente del tipo de gestión que se haga de los mensajes, es decir, se gestionen mediante el Process Data Objetcs o el Service Data Objects, los mensajes siguen una misma estructura.

Se utilizará el estándar 2.0A de CANBus, con identificadores reducidos de 11bits. El mensaje CAN completo constara de 130 bits máximos, donde se incluyen los64 bits máximos disponibles para datos.

Independientemente de la tarjeta de red que se utilice, los únicos campos del mensaje que tenemos que facilitar para que se construya el mensaje completo son el identificador y los datos.

El identificador correspondiente a los mensajes de inicialización es el 2XX, donde XX corresponde al identificador del nodo.

El identificador necesario para los demás mensajes de configuración y ejecución es 6XX. Este identificador es gestionado por el service data objects (SDO). El SDO accede a las direcciones (índice y subíndice) de los objetos predefinidos en el software. Estos objetos son equiparables a las funciones de cualquier programa.

Los objetos definidos para la gestión de la red se incluyen en los chips que incluyen el protocolo CANBus, independientemente se trate de maestro o esclavo. Existen objetos adicionales definidos por cada marca, como es el caso de los drivers de los motores, que posibilitan el control de motores (velocidad, posición, aceleración...).

Para el caso concreto de los drivers, su protocolo interno exige recibir mensajes con unos datos concretos para poder controlar los motores o para mandar mensajes de respuesta con la información solicitada. Los datos necesarios para construir el mensaje son:

- COB-ID: el COB-ID es el Identificador del Campo de Arbitraje de un mensaje CAN. Este campo recoge los 11 bits del identificador y depende directamente del tipo de mensaje a enviar a través del bus de datos. Los 4 más significativos (del 8 al 11) corresponden al tipo de objeto de comunicación y los 7 menos significativos a la dirección del dispositivo. Por lo tanto se pueden enviar 16 tipos de mensajes, y se pueden colocar en la red hasta 127 dispositivos. La tabla 8 recoge los distintos valores del COB-ID según el mensaje a enviar:

Tabla 8: Identificadores de comunicación

COB Type	Bits 8 - 11 of COB-ID	ID Range
NMT	0000	0
SYNC	0001	128 (80h)
Time Stamp	0010	256 (100h)
Emergency	0001	129...255 (81h...ffh)
PDO1 - Transmit	0011	385...511 (181h...1ffh)
PDO1 - Receive	0100	513...639 (201h...27fh)
PDO2 - Transmit	0101	641...767 (281h...2ffh)
PDO2 - Receive	0110	769...895 (301h...37fh)
PDO3 - Transmit	0111	897...1023 (381h...3ffh)
PDO3 - Receive	1000	1025...1151 (401h...47fh)
PDO4 - Transmit	1001	1153...1279 (481h...4ffh)
PDO4 - Receive	1010	1281...1407 (501h...57fh)
SDO - Transmit	1011	1409...1535 (581h...5ffh)
SDO - Receive	1100	1537...1663 (601h...67fh)
Error control (node guarding)	1110	1793...1919 (701h...77fh)

- Tamaño de datos: la tabla 9 indica el tamaño en bits que son datos útiles, es decir parámetros.

Tabla 9: Tamaño de datos

Tamaño	Valor
0 *	40h
8 bits	2Fh
16 bits	2Bh
32 bits	23h

* son mensajes de solo lectura y aquellos que devuelven un mensaje con datos concretos.

- Índice: dirección del objeto con el que se quiere comunicar. Consta de 16 bits.

- Subíndice: en cada objeto pueden existir varias "mini funciones", identificadas por su correspondiente subíndice de 8 bits. En caso de no existir subíndice en el objeto seria 00h.

- Parámetros: son los datos que necesita el objeto. Su tamaño varía en función del tipo de dato que maneje el objeto. Si un objeto no los necesitase o no usase los 32 bits máximos disponibles, por ejemplo objetos de solo lectura, se completaría con todo '0'.

Se puede observar un ejemplo en la figura 34

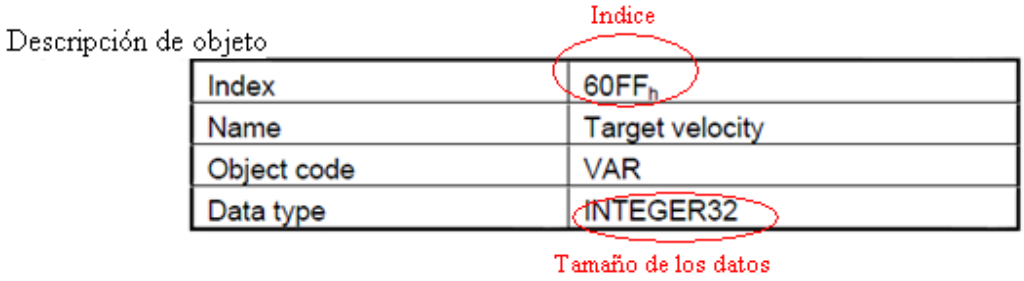


Figura 34: Descripción de objeto

- El tamaño es INTEGER32, se coge el número, en este caso el 32, y en la tabla 9 (tamaño de datos) se explican los valores asignados para cada caso, por tanto 23h.

- El índice es 60FFh, el formato de lectura es del bit menos significativo al más significativo, por tanto seria FF 60h.

- No tiene subíndice, por tanto 00h.

- Para un encoder de 500 líneas y conseguir una velocidad de 600 rpm el parámetro que hay que introducir seria 00000F00h (se explica en el apartado 3), pero hay que escribirlo del bit menos significativo al más significativo, por tanto 000F0000h.

El resultado sería el siguiente:

Tamaño de datos	Índice	Subíndice	Parámetros
23	FF 60	00	00 0F 00 00

Entonces se enviara los datos"0x23,0xFF,0x60,0x00,0x00,0x0F,0x00,0x00".
El 0xXX indica que el número está en hexadecimal.

3.2.7 Diccionario de objetos

La parte más importante del perfil de un dispositivo es la descripción del Diccionario de Objetos [ver referencia 4] que es un grupo de objetos accesibles de forma predefinida a través del bus mediante los cuales se realiza la comunicación con el dispositivo. Los objetos del diccionario están direccionados mediante un índice de 16 bits y un subíndice de 8 bits, que permiten un máximo de 65536 entradas. Su estructura es similar a la de una estructura en lenguaje C.

En el Diccionario de objetos, algunas entradas son comunes a todos los dispositivos y otras son específicas.

En este documento no se hará una descripción individual de todos los Objeto del Diccionario ya que esto supondría entretenerse mucho, además, esta información puede consultarse en el documento que se indica en la biografía (CANOpen Programming Technosoft).

En CANOpen se define para cada objeto del diccionario su función, nombre, índice, subíndice, tipos de datos, si es obligatorio u opcional, si es de "solo lectura", "solo escritura" o "lectura y escritura", etc. En la tabla 10 se puede observar el objeto RPDO1 (1400h).

Tabla 10: Diccionario de objetos

Object description:

Indice

Index	1400_h
Name	RPDO1 Communication Parameter
Object code	RECORD
Data type	PDO CommPar

Entry description:

Subindice

Sub-index	00_h
Description	Number of entries
Access	RO
PDO mapping	No
Value range	-
Default value	2

Tamaño de datos

Sub-index	01_h
Description	COB-ID RPDO1
Access	RW
PDO mapping	No
Value range	UNSIGNED32
Default value	200_h + Node-ID

Sub-index	02_h
Description	Transmission type
Access	RW
PDO mapping	No

3.2.8 Unidades de control y estado

Existen diferentes objetos para el control del actuador y la verificación de los procesos realizados. Ellos son:

- Control Word (palabra de control): cambia los estados del dispositivo.
- Status Word (palabra de estado): verifica el estado actual del driver.
- Control Mode (modo de control): cambia los modos de operación del driver

3.2.8.1 Control Word

Es el objeto de índice 6040h, que se muestra en la tabla 11, se puede controlar la máquina de estados del driver. Esto se utiliza para habilitar/deshabilitar su alimentación, comenzar/parar los movimientos y para sacarlo del estado de fallo. La Palabra de Control consta de 16 bits y la siguiente tabla muestra el significado de cada uno de ellos:

Tabla 11: Control Word

Bit	Valor	Descripción
15	0	Modo de registro inactivo
	1	Modo de registro activo
14	1	Cuando se realiza una actualización, no se actualizan los valores de velocidad y posición
13		Cuando está a 1 cancela la ejecución de la función de TML llamada por el objeto 2006h. El bit automáticamente es reinicializado por el driver cuando se ejecuta la orden.
12	0	Sin acción
	1	Si el bit 14=1 pone la posición demandada a 0
		Si el bit 14=0 pone la posición actual a 0
11		El significado de este bit depende del modo de funcionamiento del driver
10, 9		Reservado
8	0	Sin acción
	1	El motor frena
7	0	Sin acción
	1	Reseteo de los fallos. En la transición de 1 a 0 de este bit se resetean los fallos.
4,5,6		Modo de operación
3		Habilitar operación
2		Parada rápida
1		Habilitación del voltaje
0		Encendido

3.2.8.2 Status Word

Con el objeto de índice 6041h, que se muestra en la tabla 12, se pueden consultar el estado actual en que se encuentra el driver.

Tabla 12: Status Word

Bit	Valor	Descripción
15	0	Eje apagado. Etapa de potencia desactivada. No se realiza control del motor
	1	Eje encendido. Etapa de potencia desactivada .Se realiza control del motor
14	1	El último evento ha ocurrido
13, 12		Modo de operación específica.
11		Límite interno activado
10		Objetivo alcanzado
9	0	Remoto- el driver esta en modo local y no se ejecutara el mensaje
	1	Remoto- los parámetros del driver pueden ser modificados por via CAN
8	0	Ninguna función TML se ejecuta. La ejecución de la ultima función TML se ha completado
	1	Una función TML se ejecuta. No se puede llamar a otra función hasta que termine o se cancele la anterior.
7	0	Sin peligro
	1	Alerta. Una función TML es llamada mientras todavia sigue otra función en ejecución. La última función es ignorada.
6		Encendido deshabilitado
5		Parada rápida. Si esta a 0 el driver esta realizando una parada rápida
4	0	La tensión de alimentación del motor esta presente
	1	La tensión de alimentación del motor esta ausente
3		Fallo
2		Operación habilitada

3.2.8.3 Control Mode

El objeto de índice 6060h, que se muestra en la tabla 13, es un objeto de solo escritura por lo que no se puede leer de él. Según el valor que se escribe el driver entrará en un modo de control u otro:

Tabla 13: Modos de control

Valor	Descripción
-128 hasta el -6	Reservado
-5	Modo de momento de rotación de referencia externo
-3	Modo de posición de referencia externo
-2	Modo de posición ECAM
-1	Modo de posición EG
0	Reservado
1	Modo de perfil de posición
2	Reservado
3	Modo de perfil de velocidad
4,5	Reservado
6	Modo homing
7	Modo de posición interpolado
8 hasta el 127	Reservado

3.2.9 Proceso de inicialización

Para poder inicializar los driver hay que seguir unos pasos, que están señalados en la figura 35.

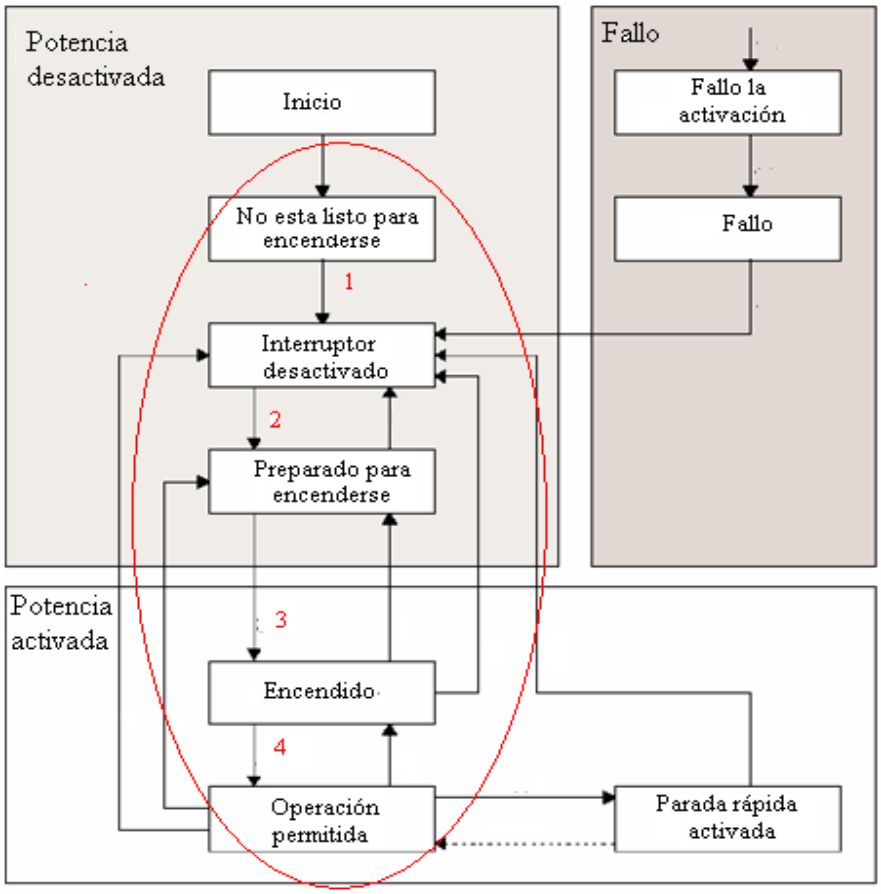


Figura 35: Proceso de inicialización

Se deben inicializar todos los nodos, así cada nodo queda activo y configurado para recibir las diferentes consultas. Se va a realizar la inicialización del nodo 1, los demás nodos serian igual con las excepción del id del nodo deseado.

1. Iniciar el nodo 1con un mensaje NMT. La estado sería interruptor desactivado

Cob- ID	Parámetros
0	01 01

2. Cambia el estado del driver a preparado para encenderse.

Cob- ID	Tamaño de datos	Índice	Subíndice	Parámetros
601h(600 +1 del nodo)	2B	40 60	00	06 00 00 00

3. Para cambiar el estado del driver ha encendido.

Cob- ID	Tamaño de datos	Índice	Subíndice	Parámetros
601h(600 +1 del nodo)	2B	40 60	00	07 00 00 00

4. Para cambiar el estado a operación permitida.

Cob- ID	Tamaño de datos	Índice	Subíndice	Parámetros
601h(600 +1 del nodo)	2B	40 60	00	0F 00 00 00

Estos tres últimos mensajes también podrían haberse enviado a través de PDOs.

Hasta aquí es la inicialización, ahora se realizara un ejemplo de posicionamiento del motor en un ángulo de 45 grados con una velocidad de 600rpm.

5. Se selecciona el modo de operación deseado, en este caso, modo de perfil de posición. En la tabla número 13 se puede observar que es el valor 1.

Cob- ID	Tamaño de datos	Índice	Subíndice	Parámetros
601h(600 +1 del nodo)	2F	60 60	00	01 00 00 00

6. Se envía la posición deseada. En este caso 4 rotaciones, teniendo en cuenta que se usa un encoder de 500 líneas, el valor sería 1F40h.

Cob- ID	Tamaño de datos	Índice	Subíndice	Parámetros
601h(600 +1 del nodo)	23	7A 60	00	401F 00 00

7. Se envía la velocidad. En este caso 500 rpm, haciendo la cuenta seria 10AAACh.

Cob- ID	Tamaño de datos	Índice	Subíndice	Parámetros
601h(600 +1 del nodo)	23	81 60	00	AC AA 10 00

8. Arrancar.

Cob- ID	Parámetros
201h(200 +1 del nodo)	1F 00

9. Resetear.

Cob- ID	Parámetros
201h(200 +1 del nodo)	0F 00

3.3 Herramienta de simulación y diseño Matlab.

3.3.1 Introducción

Matlab es la abreviatura de MATrix LABoratory (laboratorio de matrices) es un software matemático que ofrece un entorno de desarrollo integrado (IDE) con un lenguaje de programación propio (lenguaje M). Está disponible para las plataformas Unix, Windows y Apple Mac OS X.
Entre sus prestaciones básicas se hallan: la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos, la creación de interfaces de usuario (GUI) y la comunicación con programas en otros lenguajes y con otros dispositivos hardware. El paquete Matlab dispone de dos herramientas adicionales que expanden sus prestaciones, a saber, Simulink (plataforma de simulación multidominio) y GUIDE (editor de interfaces de usuario - GUI). Además, se pueden ampliar las capacidades de MATLAB con las cajas de herramientas (toolboxes); y las de Simulink con los paquetes de bloques (blocksets).
Es un software muy usado en universidades y centros de investigación y desarrollo. En los últimos años ha aumentado el número de prestaciones, como la de programar directamente procesadores digitales de señal o crear código VHDL.

3.3.2 Realización de una GUIDE

GUIDE es un entorno de programación visual disponible en Matlab para realizar y ejecutar programas que necesiten ingreso continuo de datos. Tiene las características básicas de todos los programas visuales como Visual Basic o Visual C++.
Para crear una interfaz, lo primero que hay que hacer es abrir la aplicación, se puede escribir en la consola de Matlab directamente la palabra “guide” o bien se puede pulsar el icono que muestra la figura 36.

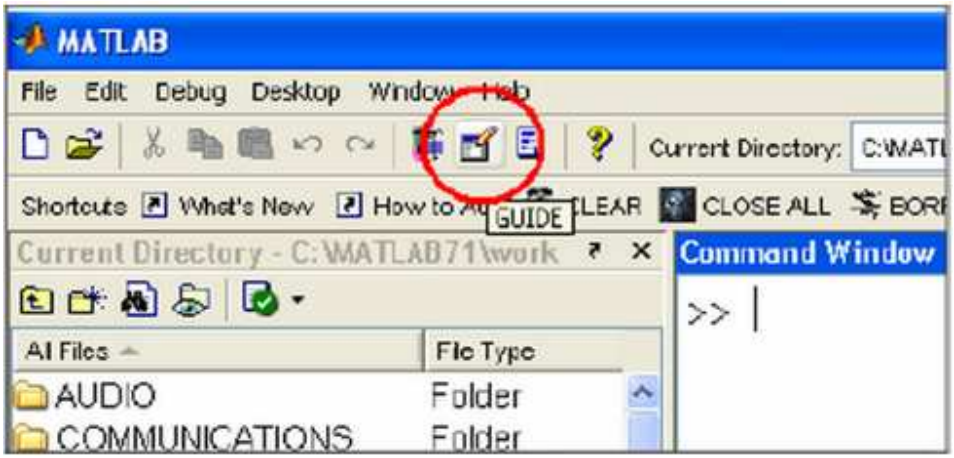


Figura 36: Icono GUI

Si todo es correcto, se muestra la figura 37. En ella se pulsa la primera opción “Blank Gui”.

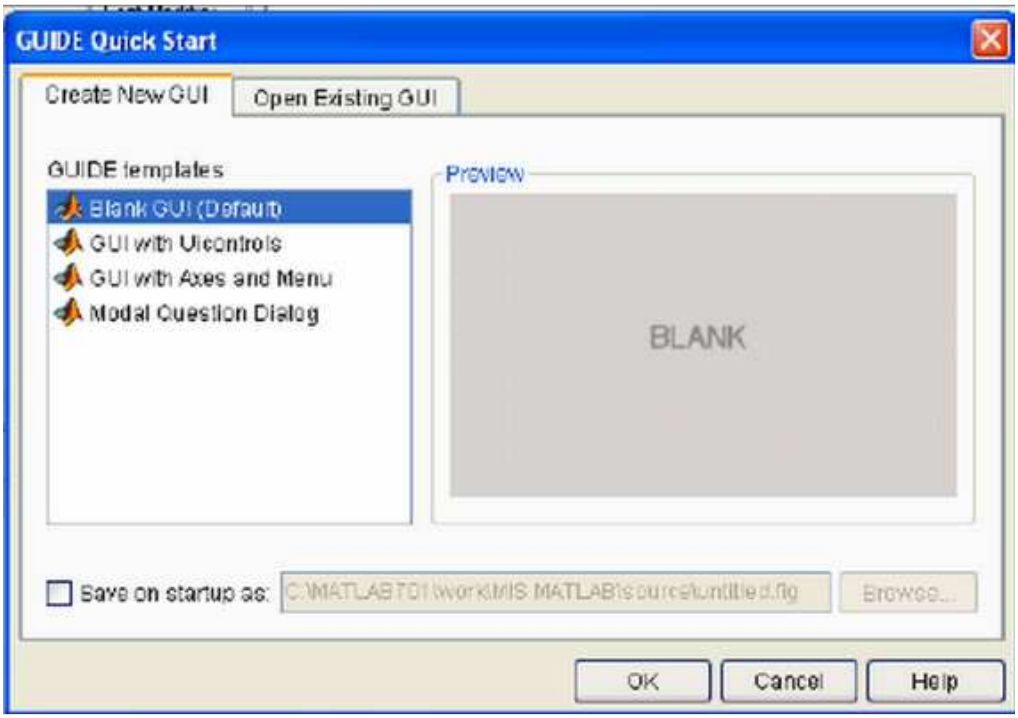


Figura 37: Ventana de inicio GUI

Representa un formulario en blanco para poder realizar la interfaz como muestra la figura 38. Se pueden diferenciar tres zonas:

- Área de diseño: donde se compone la interfaz.
- Paleta de componentes: que expone las diferentes opciones que se pueden realizar.
- Herramientas: para realizar ajustes y cambios en los componentes.

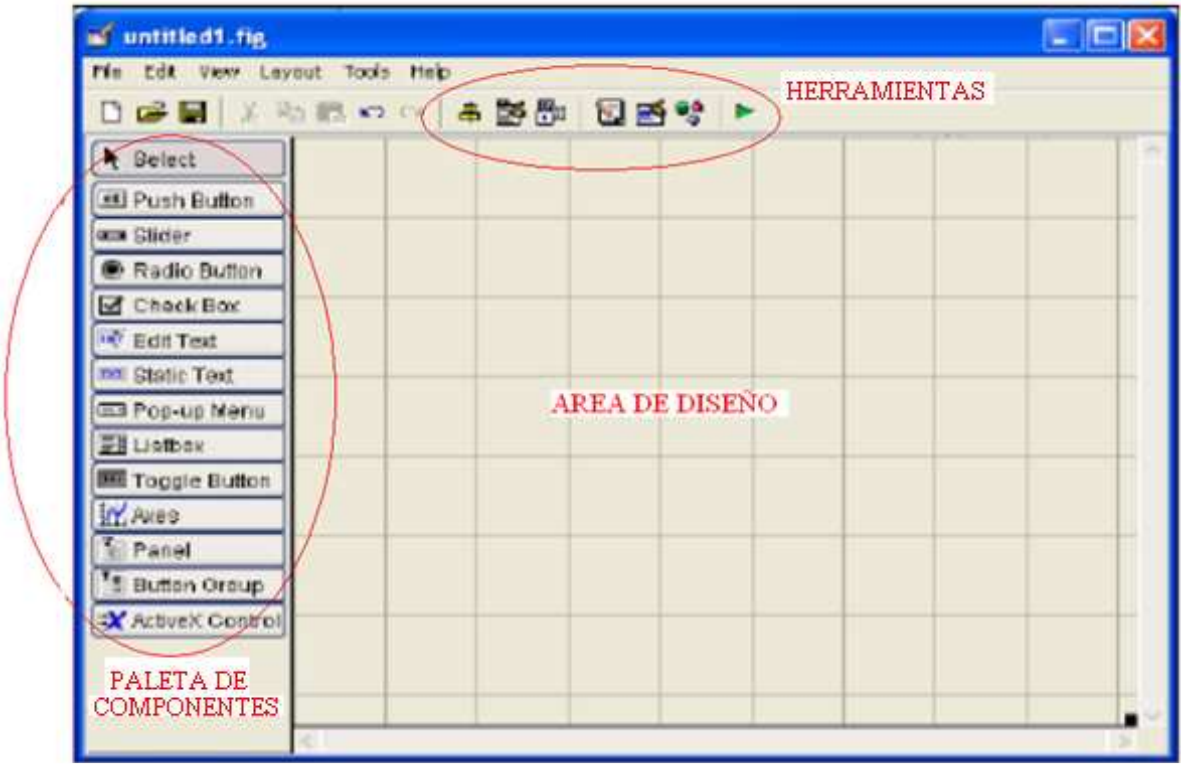


Figura 38: Diseño GUI

Las herramientas son las siguientes:

Alinear objetos.

Editor de menú.

Editor de orden de etiqueta.

Editor del M-file.

Propiedades de objetos.

Navegador de objetos.

Grabar y ejecutar.

La descripción de la paleta de componentes se muestra en la tabla 14.

Tabla 14: Descripción componentes

Control	Valor de estilo	Descripción
Check box	'checkbox'	Indica el estado de una opción o atributo
Editable Text	'edit'	Caja para editar texto
Pop-up menu	'popupmenu'	Provee una lista de opciones
List Box	'listbox'	Muestra una lista deslizable
Push Button	'pushbutton'	Invoca un evento inmediatamente
Radio Button	'radio'	Indica una opción que puede ser seleccionada
Toggle Button	'togglebutton'	Solo dos estados, "on" o "off"
Slider	'slider'	Usado para representar un rango de valores
Static Text	'text'	Muestra un string de texto en una caja
Panel button		Agrupa botones como un grupo
Button Group		Permite exclusividad de selección con los radio button
Axes		Realiza Graficas

Las interfaces resultantes se proceden a explicar en el capítulo 4.4.3.

3.3.3 Uso de archivos MEX

Los archivos llamados "Mex-files" son funciones en lenguaje C que pueden ser llamadas en Matlab como una función propia. De este modo, una función *.m de Matlab puede ser sustituida por una función programada en C que se llama exactamente de la misma forma. Estas funciones se compilan y generan librerías compartidas que son las denominadas funciones MEX.

Las funciones MEX tienen una extensión diferente en función de los sistemas operativos en que hayan sido generadas, en este caso, la extensión es .mexglx, ya que se utiliza Linux.

El código fuente de un fichero MEX programado en C tiene dos partes. La primera parte contiene el código de la función C que se quiere implementar como fichero MEX. La segunda parte es la función mexFunction que hace de interface entre C y Matlab

Los ficheros MEX deben incluir la librería "mex.h" donde está declarada la función mexFunction. El encabezado de la función debe tener la siguiente forma:

void mexFunction (int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])

- nlhs: el número de argumentos esperados en la salida.
- plhs: es un vector de punteros a los valores de los argumentos de salida que va a pasar la función C. Serán los parámetros que el programa introduzca en Matlab.
- nrhs: el número de argumentos de entrada.

- prhs: es un vector de punteros a los valores de los argumentos de entrada (‘right hand side arguments’) que se van a pasar a la función C. Serán los parámetros que Matlab introduzca en el programa.

Antes de seguir adelante con esta explicación es conveniente decir algo sobre los mxArrays, que son los únicos objetos con los que trabaja Matlab. Todos los tipos de variables de Matlab (escalares, vectores, matrices, cadenas de caracteres, estructuras, vectores de celdas, etc.) son mxArrays. Para cada mxArray, Matlab almacena el tipo, las dimensiones, los datos, si es real o complejo (para datos numéricos), el número de campos y sus nombres para las estructuras, etc. Matlab dispone de un gran número de funciones C para trabajar con mxArrays, que pueden encontrarse buscando ‘MX Array Manipulation (C)’ en el ‘Help’.

Estos mxArrays son de sólo lectura y no debe ser modificado por el MEX-file. La modificación de los datos en estos mxArrays puede producir efectos secundarios no deseados.

En la figura 39 se muestra la conexión entre C y Matlab, cómo llegan los datos al fichero MEX, qué es lo que se hace con estos datos en mex-Function y cómo se devuelven finalmente los resultados a MATLAB.

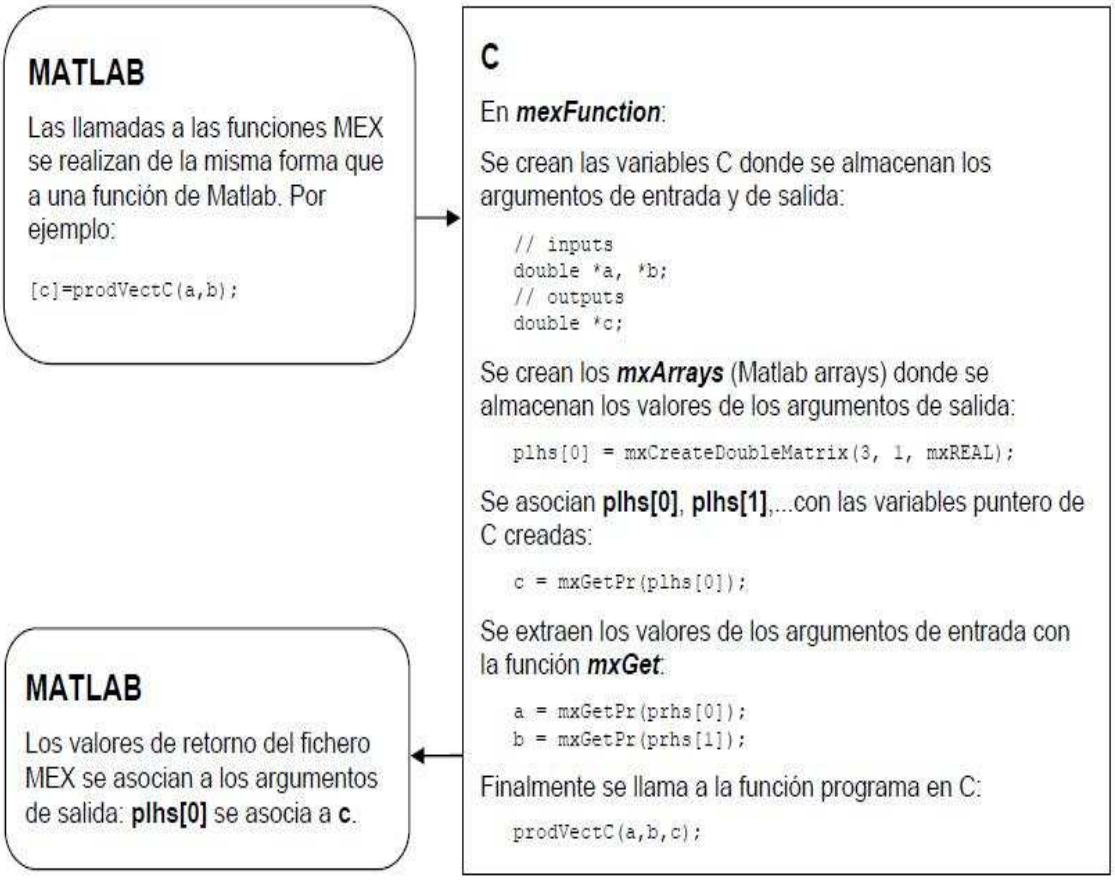


Figura 39: Esquema creación de una función MEX

CAPITULO 4: DESARROLLO DE LA APLICACIÓN DE CONTROL DIRECTO DE EJES

4.1 Introducción

El esquema hardware se muestra en la figura 40:

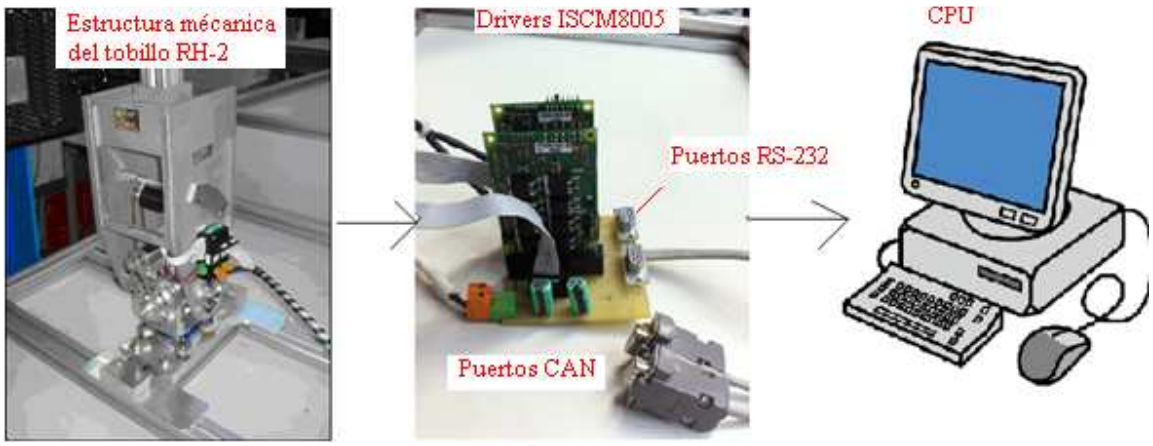


Figura 40: Esquema de la aplicación

La unión del PC con el driver puede ser por medio de RS232 si se quiere configurar el driver o bien por medio de CAN cuando se desea ejecutar el programa.

4.2 Configuración software de los drivers ISCM8005

Para un correcto funcionamiento del driver hay que realizar diversos ajustes:

- Cambio de firmware: permite activar la conexión CANOpen.
- Configuración del motor: Cambia los parámetros del motor y realiza los test de comprobación.

4.2.1 Cambio de Firmware

Para poder cambiar el firmware, habilita la comunicación por el puerto, hay que conectar el driver al puerto serie RS-232 del PC. Hecho esto se debe ejecutar el programa Firmware Programmer que está incluido en el software del driver, figura 41.

Se observa lo siguiente:

1. Cuando se haya abierto se pulsa en el botón "check comunication" y se comprueba la conexión con el driver y la versión del firmware que lleva.

2. En la opción "select firmware" hay que seleccionar F250C ya que esto habilita la comunicación por el puerto.

3. Se finaliza el proceso pulsando "start programing". Nunca hay que desconectar la alimentación del driver mientras dure el proceso de programación ya que puede estropearse.

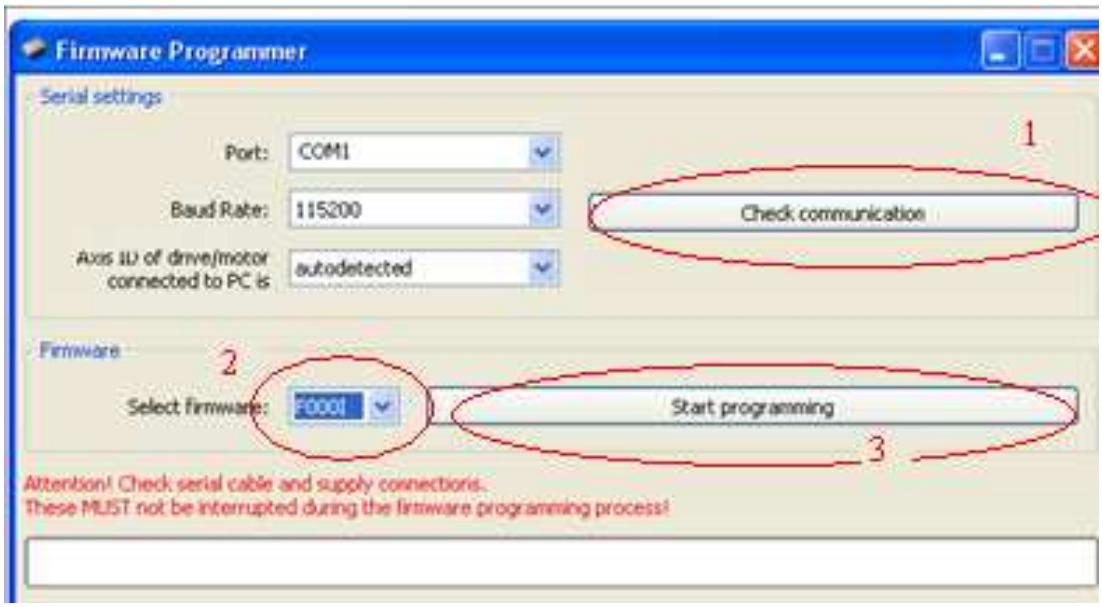


Figura 41: Programar firmware

4.2.2 Configuración del motor

Cuando se conecta un nuevo motor al driver se deben obtener las características del mismo: intensidad, número de líneas del enconder, etc. Para ello se abre la aplicación EasyMotion Studio, figura 42, incluida en el software del driver. Una vez abierto se crea un nuevo proyecto o se abre uno creado previamente.

Se pulsa el botón New para abrir "New Project" y seleccione el tipo de unidad: ISCM8005; el tipo de motor: brushed, brusless,...; el control de modo de bucle cerrado o abierto y tipo de dispositivo de retroalimentación [ver referencia 5].

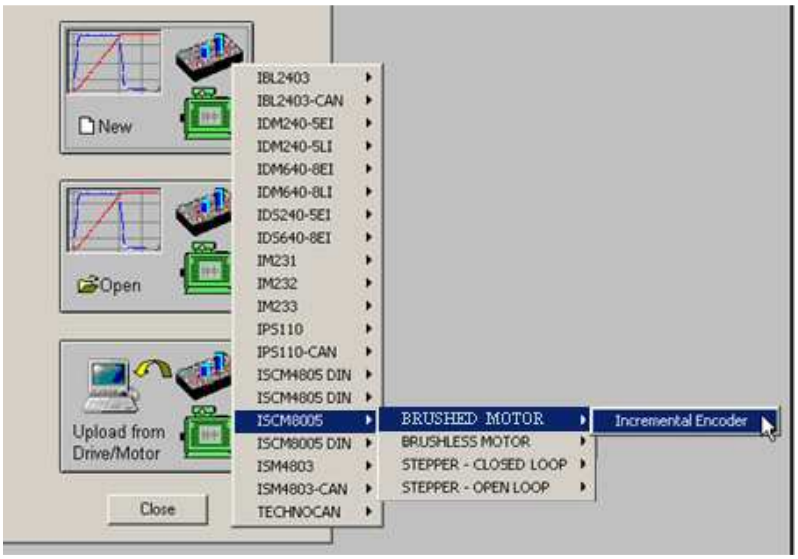


Figura 42: Elección del driver

Si se pulsa la opción Setup aparecerá una serie de opciones desde las cuales se podrá crear una nueva configuración del driver, cargar una ya creada, descargarla al driver, etc. Como se observa en la figura 43.

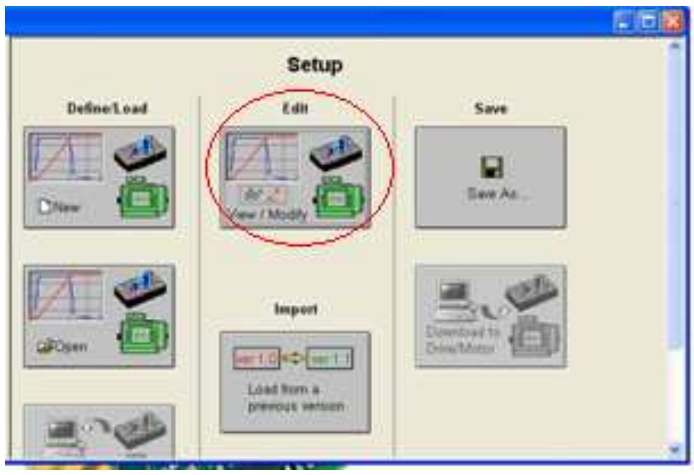


Figura 43: Setup EasyMotion

Para crear una nueva configuración hay que pulsar en Edit, tras lo cual aparecerán dos nuevas ventanas, una para configurar los parámetros del motor y otra para los parámetros del driver

4.2.3. Parámetros del motor

Aquí ya se puede configurar los parámetros del motor, figura 44. Según el motor que se haya elegido en el primer paso algunas de las opciones pueden estar o no deshabilitadas.

Ahora hay que configurar el motor, para ello:

1. Se pulsa en Database para seleccionar el fabricante y el modelo.

2. Si el motor no se encuentra en esta base de datos se elige la opción User y se configura manualmente según los datos de las hojas de características.

3. Indicar el número de líneas que tiene el encoder (en este caso 500 según la tabla 5)y realizar el "test connections" para comprobar el buen funcionamiento.

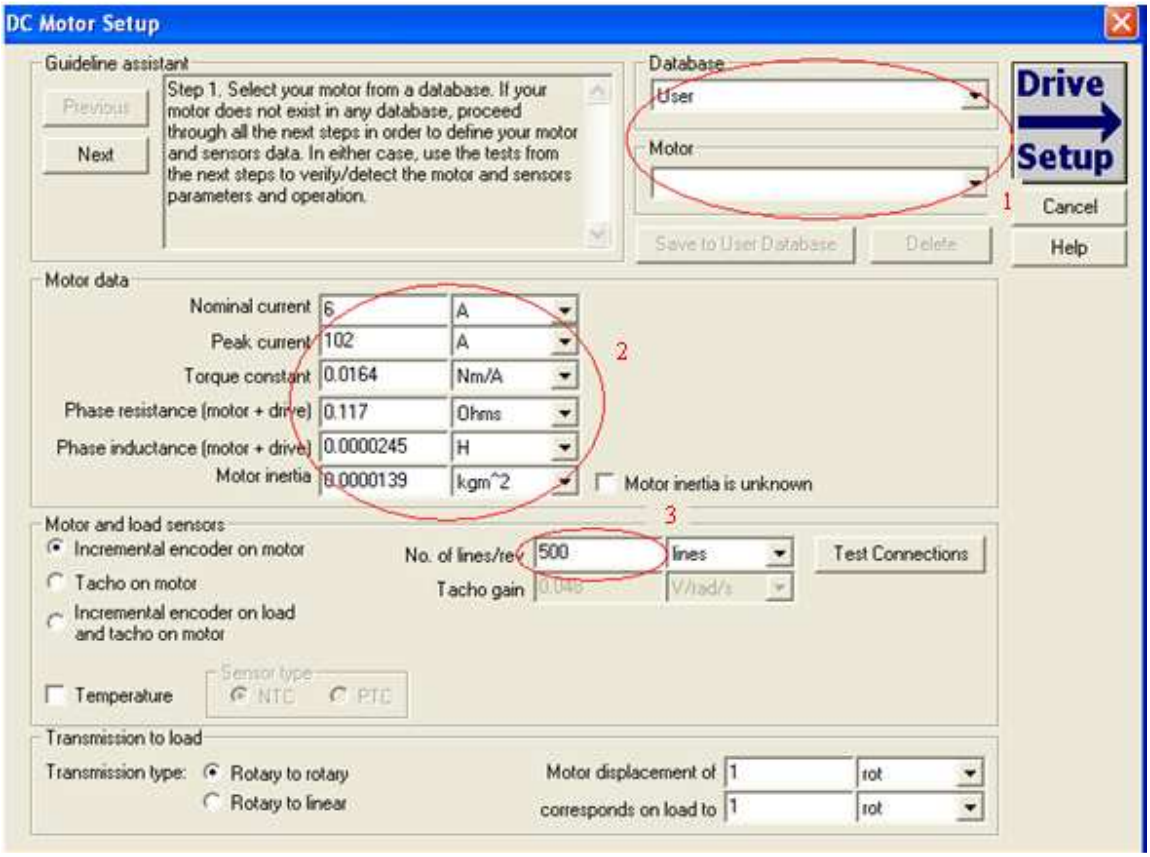


Figura 44: Motor setup

4.2.4. Parámetros del driver

En la configuración del driver, que se muestra en la figura 45, aparecen las siguientes opciones:

1. Elección del número del driver, en nuestro caso el driver se selecciona por hardware ("set/change axisID").

2. Modo de control, indica la acción que quiere realizar el driver, en nuestro caso mover en posición ("control mode").

3. Velocidad del CANBus: se debe poner la misma velocidad que la que tenga el programa de control hecho en C ("Baud rate").

4. Realizar los diferentes test para comprobar que el funcionamiento es correcto ("Tune & Test").

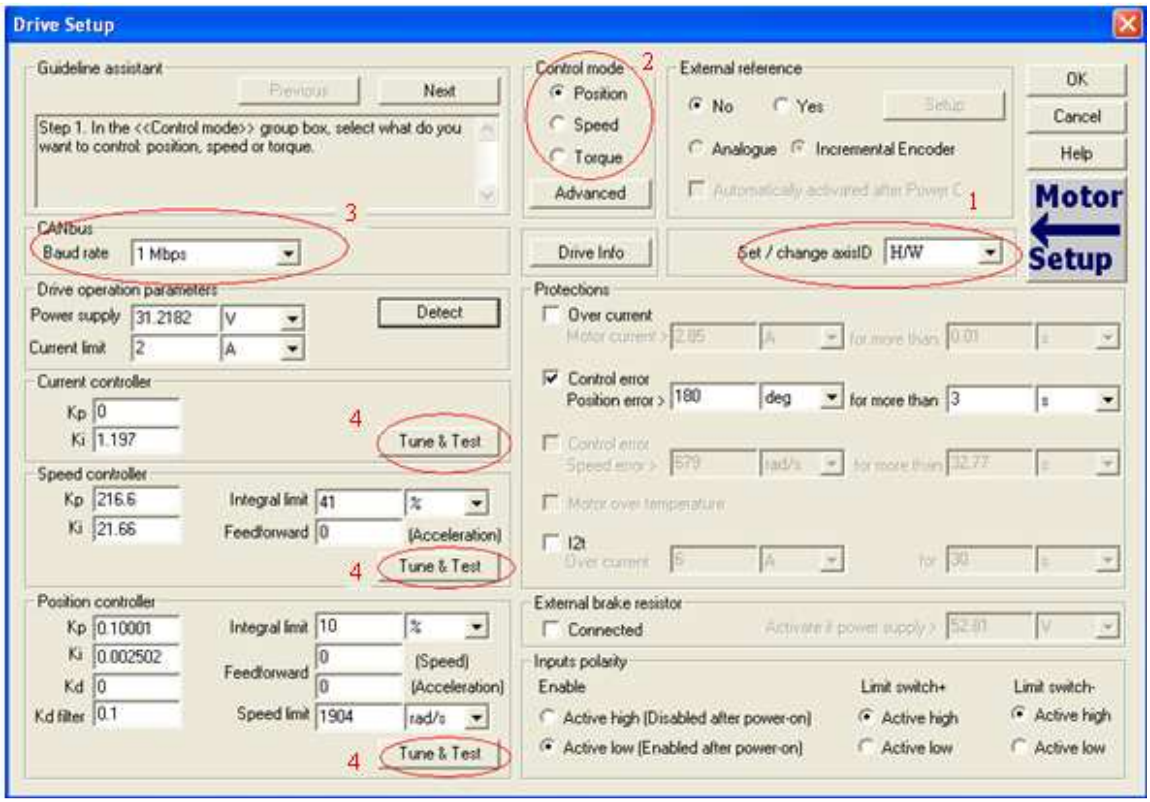


Figura 45: Driver setup

4.3 Configuración de los elemento de comunicación del driver

En el driver se usaran los siguientes los siguientes objetos de comunicación:

El PDO1-Receive, que es de la forma 200+id nodo (del 201h al 223h), por defecto viene mapeados ya al objeto 6040 (Control Word punto 2.3.8.1).

El SDO-Receive, que es de la forma 600+id nodo (601h al 623h), que será el encargado del resto de instrucciones.

4.4 Arquitectura software de la aplicación

La arquitectura software estará compuesta por la configuración del driver que se ha visto en el apartado anterior y la configuración de los programas que ejecutara el PC. Dentro de este existen, como se puede apreciar en la figura 46, tres grandes módulos:

- Módulo principal: es el encargado de la comunicación CAN y a su vez del control de los motores. A su vez, está formado por diferentes funciones, todas ellas están en la librería “cabecera.h”, que son:

    - Inicialización: inicializa los drivers usados.
    - Conversión: convierte el número para que pueda adaptarse a la trama CAN
    - Leer: recibe los valores reales del driver (posición y velocidad actual).
    - Display: muestra por pantalla los diferentes mensajes CAN.

- Módulo de enlace: se encarga de la conexión entre en el programa principal y la interfaz. Consta de tres programas:

    - Matlaenlace: envía los datos posición y velocidad de la interfaz al programa principal.
    - Matlaver: envía los datos de posición y velocidad del programa principal a la interfaz.
    - Matlanodo: envía el número de identificador de la interfaz al programa principal.

- Módulo Matlab: será la encargada de la conexión con el usuario.

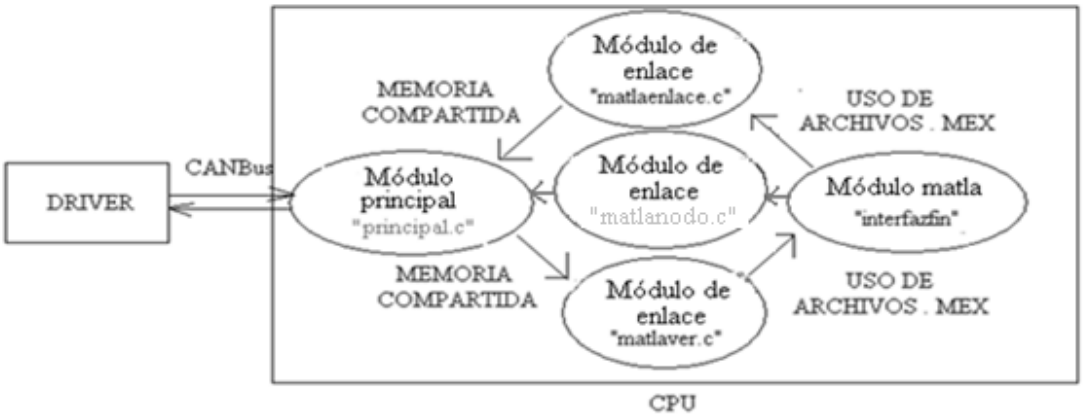


Figura 46: Módulo software

4.4.1 Módulo principal

Como se ha explicado en el apartado anterior, este programa será el encargado de mandar y recibir del driver los diferentes mensajes CAN y a su vez, enviar y recibir datos de los demás programas del sistema.

Tiene el diagrama de flujo que representa la figura 47 y que se explica después de la figura

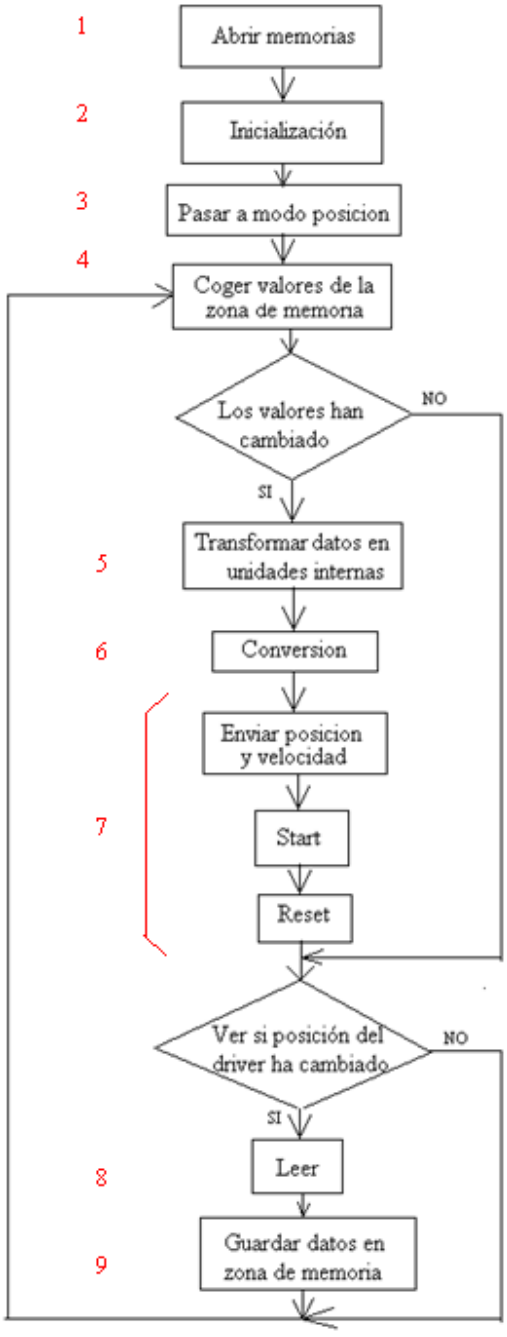


Figura 47: Flujograma módulo principal

Este programa, llamado principal.c, está escrito en lenguaje de programación C y para un sistema operativo Linux. Consta de una cabecera donde se escribirán todas las funciones que utiliza dicho programa (cabecera.h).

El programa, como se observa en la figura 47, sigue los siguientes pasos:

1. Abre las zonas de memoria.

2. Llama a la función inicialización.

3. Pone los drivers en modo posición.

```
uint8_t buf6[8] = {0x2F,0x60,0x60,0x00,0x01,0x00,0x00,0x00};
for (idmotor=1;idmotor<25;idmotor++)
      {  printf("modo\n");
         mensaje.id=0x600+idmotor
                  mensaje.dlc=8;
         memcpy(mensaje.data,buf6,8*sizeof(uint8_t));
write(nodoTx, &mensaje, sizeof(struct can_msg));
sleep(0.05); }
```

   La variable buf6[8] son los datos a enviar que se introduce en la parte del mensaje correspondiente a los datos (mensajse.data) [ver referencia 6], se guardan también el identificador del nodo correspondiente (mensaje.id) y el tamaño (tamaño.dlc). Por último se manda el mensaje por el nodoTx, se realiza una pausa para que el mensaje sea enviado correctamente.

4. Coge los valores de la zona de memoria.

5. Convierte los números de revoluciones y grados en unidades internas del sistema.

   - Para grados se usa la ecuación 1

$$Pui = \frac{Pg \times 4 \times N \times Tr}{360} \quad (1)$$

   Dónde: Pui: la variable donde se almacena.
   Pg: los grados a convertir.
   N: número de líneas del encoder (en este caso 500).
   Tr: la relación de transformación (en este caso 320).

   - Para rpm se usa la ecuación 2

$$Vui = \frac{Vrpm \times 4 \times N \times Tr \times T}{60} \quad (2)$$

Dónde: Vui: la variable donde se almacena.
Vrpm: las rpm a convertir.
N: número de líneas del encoder (en este caso 500).
Tr: la relación de transformación (en este caso 320).
T: 0.001(1ms).

6. Llama a la función conversión.

7. Envíala posición y velocidad al driver deseado según los datos que le envíe el programa de enlace (matlaenlace.c), utilizando los objetos 6081h y 607Ah.

```
uint8_t buf7[8] = {0x23,0x7A,0x60,0x00,0x9C,0xAD,0x00,0x00};
uint8_t buf8[8] = {0x23,0x81,0x60,0x00,0x00,0x0F,0x00,0x00};
uint8_t buf9[2] = {0x1F,0x00};
uint8_t buf10[2] = {0x0F,0x00};

printf("posicion\n");
mensaje.id=0x600+idmotor;
mensaje.dlc=8;
memcpy(mensaje.data,buf7,8*sizeof(uint8_t ));
write(nodoTx, &mensaje, sizeof(struct can_msg));
sleep(0.05);

printf("velociad\n");
memcpy(mensaje.data,buf8,8*sizeof(uint8_t));
write(nodoTx, &mensaje, sizeof(struct can_msg));
sleep(0.5);

printf("start\n");
mensaje.id=0x200+idmotor
mensaje.dlc=2;
memcpy(mensaje.data,buf9,2*sizeof(uint8_t));
write(nodoTx, &mensaje, sizeof(struct can_msg));
sleep(3);

printf("reset\n");
memcpy(mensaje.data,buf10,2*sizeof(uint8_t));
write(nodoTx, &mensaje, sizeof(struct can_msg));
sleep(1);
```

La variable buf7[8] indica los 45º, buf8[8] las 600 rpm, buf9[2] start y buf10[2]

Reset. En la variable mensaje se procede a guardar los variables de identificación (menaje.id, guarda SDO+id del driver o el PDO+ id), tamaño (mensaje.dlc, guarda un 8 ó un 2 según sea preciso) y los datos (mensaje.data, guarda la variable buf que se quiera enviar en ese momento).Se realiza una pausa para que el mensaje sea enviado correctamente.

8. Llama a la función leer.

9. Guarda los datos obtenidos por la función leer en la otra zona de memoria.

Las funciones incluidas en “cabecera.h” son:

- Inicializacion.c: se encarga de abrir los nodos del puerto CAN e inicializar los driver para que se preparen para el proceso. Su diagrama de flujo es el de la figura 48.

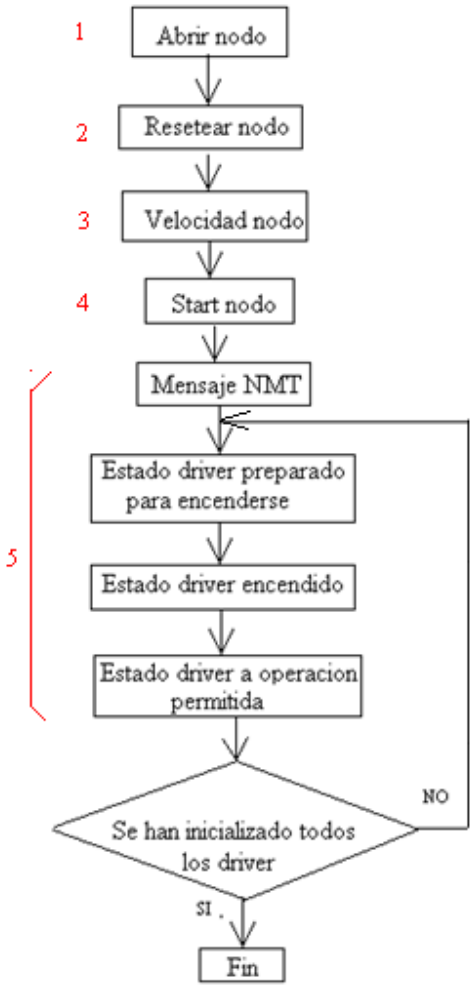


Figura 48: Flujograma inicialización

1. Abrir nodo[ver referencia 7]: nodoTx=open("/dev/can1",O_RDWR);
2. Reset: ioctl(nodoTx,IOC_RESET_BOARD);
3. Velocidad: ioctl(nodoTx,IOC_SET_BITRATE,& BITRATE_1000k);
4. Start: ioctl(nodoTx,IOC_START);
5. Una vez abierto los nodos del CAN, se sigue inicializando los drives, como se explicó en el capítulo 3.2.9.

```
uint8_t buf0[2] = {0x01,0x01};
uint8_t buf1[8] = {0x2B,0x40,0x60,0x00,0x06,0x00,0x00,0x00};
uint8_t buf2[8] = {0x2B,0x40,0x60,0x00,0x07,0x00,0x00,0x00};
uint8_t buf3[8] = {0x2B,0x40,0x60,0x00,0x0F,0x00,0x00,0x00};

mensaje.ff=FF_NORMAL;
mensaje.id=0x00;
mensaje.dlc=2;
memcpy(mensaje.data,buf0,2*sizeof(uint8_t));
write(nodoTx, &mensaje, sizeof(struct can_msg));


for (idmotor=1;idmotor<25;idmotor++)
{   //1 estado del driver preparado para encenderse
      mensaje.id=0x600+idmotor;
mensaje.dlc=8;
      memcpy(mensaje.data,buf1,8*sizeof(uint8_t));
      write(nodoTx, &mensaje, sizeof(struct can_msg));
sleep(0.05);

      // 2 estado del driver encendido
      memcpy(mensaje.data,buf2,8*sizeof(uint8_t));
      write(nodoTx, &mensaje, sizeof(struct can_msg));
      sleep(0.05);

      // 3 estado del driver a operación permitida
      memcpy(mensaje.data,buf3,8*sizeof(uint8_t));
      write(nodoTx, &mensaje, sizeof(struct can_msg));
sleep(0.05);
}
```

Se abre el puerto CAN, a partir de ahora todos los mensajes que quieras enviar deben de ser a través de este nodo (nodoTx), se reinicia el nodo, se determina la velocidad (1megabaudios) y se activa el nodo.
El buf0[2] es el mensaje NMT, el buf1[8] es estado del driver preparado para encenderse, el buf2[8] es el estado del driver encendido y el buf3[8] es el estado del driver a operación permitida

- Conversion.c: se encarga de convertir el número mandado por el programa principal (la diferente posición o velocidad en unidades internas) en el formato adecuado para poder mandarlo por un mensaje de CAN. Su diagrama de flujo es el de la figura 49.

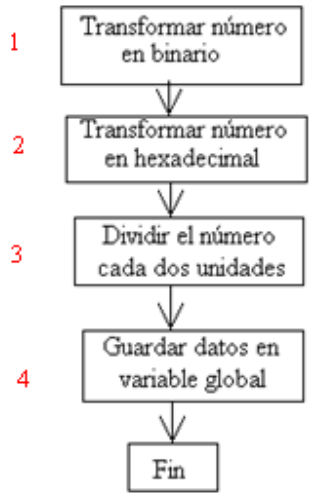


Figura 49: Flujograma conversión

1. Primero convierte en el número a binario.

```
flag=0;
if (valor<0)
{flag=1;
valor=-valor-1; }
for (i=0;i<BITS; i++)
{binNum[i]=flag^(valor%2);
valor /=2; }
```

   Para convertir el número a binario, primero hay que ver si es positivo o no, si es negativo se le resta uno, luego se hace el complemento a 2.

2. Seguidamente se convierte este número binario en un número hexadecimal.

3. Obtienes un número hexadecimal que acabas agrupando cada dos números en una variable:

```
buf[7]= hex[7]*16 +hex[6];
buf[6]= hex[5]*16 +hex[4];
buf[5]= hex[3]*16 +hex[2];
buf[4]= hex[1]*16 +hex[0];
```

   Se multiplica por 16 porque estamos en base hexadecimal.

4. Se guardan los datos de posición o velocidad en la variable global en la parte de datos.

- Leer.c: es la función encargada de obtener los valores reales de posición y velocidad del driver. Su diagrama de flujo es el de la figura 50.

  Está formada por tres funciones:

  -leer_rpm: encargada de leer las revoluciones

  -leer_grados: encargada de leer los grados

  -read_timeout: encargada de ver si hay conexión con el driver

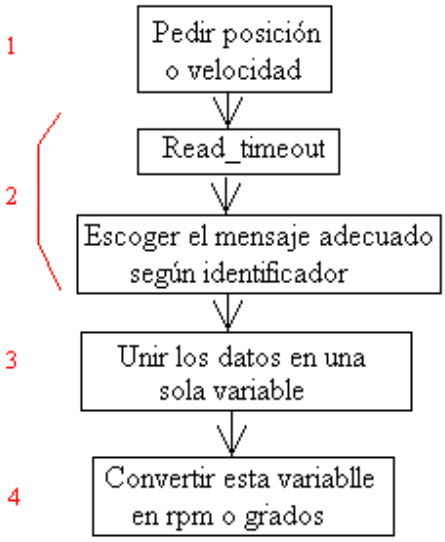


Figura 50: Flujograma leer

Para ello se usa los siguientes objetos 606C (velocidad actual) y 6064 (posición actual) son los encargados de pedir información

Para explicarlo mejor se realiza el ejemplo de pedir velocidad actual

Cob- ID	Tamaño de datos	Índice	Subíndice	Parámetros
601h(600 +nº del nodo)	40	6C60	00	00 00 00 00

Hay que seguir los siguientes pasos:

1. Mandar mensaje pidiendo velocidad.

```
uint8_t buf14[8]= {0x40,0x6C,0x60,0x00,0x00,0x00,0x00,0x00};
mensaje.id=0x600+idmotor;
mensaje.dlc=8;
memcpy(mensaje.data,buf14,8*sizeof(uint8_t));
write(nodoTx, &mensaje, sizeof(struct can_msg));
```

La variable buf14[8] es el objeto 606C que pide la velocidad actual al driver

2. Llama a la función read_timeout para comprobar la conexión y ver los mensajes recibidos. De los mensajes recibidos se busca el que tenga el identificador correspondiente, en este caso 6C 60 (al revés, ya que van del bit menos significativo al más significativo).

```
while(i== 1)
{ read_timeout(nodoTx,&mensaje_rec,2000);
if (mensaje_rec.data[1]== 108)
                    { if (mensaje_rec.data[2]== 96)
                       i=0;}
sleep(0.1); }
```

El 108 es el 6C hexadecimal y el 96 es el 60, cuando encuentra el mensaje correcto se sale del bucle (i=0)

3. La parte de los datos de ese mensaje se convierte en un número entero que estaría en unidades internas.

4. Convierte el número internas en revoluciones o grados.

   - Para convertir unidades internas a grados se usa la ecuación 3.

$$Pg = \frac{Pui \times 360}{4 \times N \times Tr} \quad (3)$$

   - Para convertir unidades internas a grados se usa la ecuación 4

$$Vrpm = \frac{Vui \times 60}{4 \times N \times Tr \times T} \quad (4)$$

- Display: muestra por pantalla el mensaje CAN y nodo al que se le manda, para comprobar que es correcto lo que se quiere enviar.

```
printf("\t ID: %X\n",mensaje.id);
printf("\tData: ");
for(i=0; i < mensaje.dlc; i++)
printf("%X - ",mensaje.data[i]);
```

Mensaje.id tiene el identificador y mensaje.data muestra los datos.

También dentro de “cabecera.h” se encuentra la librería "hico_api.h" que será la encargada de las funciones relacionadas con el CANOpen como es el abrir nodos, control de velocidad, start, reset, envió y recepción del mensaje CAN.

4.4.2 Módulo de enlace

Existen tres programas de enlace (“matlaver.c”, “matlaenlace.c” y “matlanodo.c”) debido a que Matlab no puede usar la función principal como una función suya, a causa de la complejidad de esta.

Matlaenlace.c es la encargada de recibir los datos de posición y velocidad de la interfaz y enviárselos al programa principal.c que es el encargado de mover los motores, mientras que matlaver.c realizará la acción contraria. Matlanodo.c envía en identificador de la interfaz al programa principal.

Como ya se ha explicado antes, la conexión entre estos programas y el programa principal será a través de memorias compartidas, mientras que este tipo de programas son funciones que puedes llamar en Matlab gracias a la opción “mexFunction”, ver apartado3.3.3

1. Función matlaenlace.c y

   Abre la zona de memoria común con el programa principal (llamada “memo *querida”) y se encarga del envío de la posición y velocidad de Matlab al programa principal.

```
void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[]) {

int shmid;
int vquerida,pquerida;
memo *querida;

/* Creación de la zona de memoria compartida */
if((shmid = shmget(CLAVE_SHM, sizeof(memo), IPC_CREAT|0666)) == -1){
perror("shmget");
exit(EXIT_FAILURE);
}

/* Obtención del puntero a la estructura de datos compartida */
querida = (memo *)shmat(shmid,0,0);

vquerida = (int)mxGetScalar(prhs[0]);
pquerida = (int)mxGetScalar(prhs[1]);
querida->velocidad = vquerida;
querida->posicion = pquerida;
}
```

La función mxGetScalar se encargara de recoger los valores se introducen cuando se llama a la función, se guardan los valores de posición y velocidad en las zonas de memoria

2. Función matlanodo.c:

Abre la zona de memoria común con el programa principal (llamada "memo *querida") y se encarga del envío del nodo de Matlab al programa principal.

```
void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[]) {

int shmid;
int nquerida;
memo *querida;

/* Creación de la zona de memoria compartida */
if((shmid = shmget(CLAVE_SHM, sizeof(memo), IPC_CREAT|0666)) == - 1)
{ perror("shmget");
exit(EXIT_FAILURE);
 }

/* Obtención del puntero a la estructura de datos compartida */
querida = (memo *)shmat(shmid,0,0);

   nquerida = (int)mxGetScalar(prhs[0]);
   querida->driver = nquerida;
 }
```

La función mxGetScalar se encargara de recoger los valores se introducen cuando se llama a la función, se guarda el valor del nodo en la zona de memoria

3. Función matlaver.c

Abre la otra zona de memoria común con el programa principal (llamada "ver *loquehay") y se encarga del envio de datos del programa principal a Matlab, siendo *y* la variable donde se guarda la posicion y *x* donde se guarda la velocidad.

```
void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[]) {

int shmid;
double *v,*z;
double y, x;
int vquerida,pquerida;
ver *loquehay;
```

```
/* Creación de la zona de memoria compartida */
if((shmid = shmget(CLAVE_ver, sizeof(ver), IPC_CREAT|0666)) == -1){
perror("shmget");
exit(EXIT_FAILURE);
}

/* Obtención del puntero a la estructura de datos compartida */
loquehay = (ver *)shmat(shmid,0,0);

x=loquehay->velocidad;
plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
z = mxGetPr(plhs[0]);
*(z) = x; //x serán los valores que quieras enviar a la interfaz

y=loquehay->posicion;
plhs[1] = mxCreateDoubleMatrix(1,1, mxREAL);
v = mxGetPr(plhs[1]);
*(v) = y; //y serán los valores que quieras enviar a la interfaz
```

Las variables *z* y *v*son variables auxiliares, la función mxCreateDoubleMatrix es una funcion de Matlab que adapta los datos, mxGetPr(plhs[0])adapta *z* y *v* para que guarde los valores que se quieran enviar a la interfaz.

4.4.3 Módulo Matlab

La interfaz de Matlab, figura 51, será la encargada de la interacción entre el programa y el usuario, este indicará que movimiento quiere realizar [ver referencia 11].

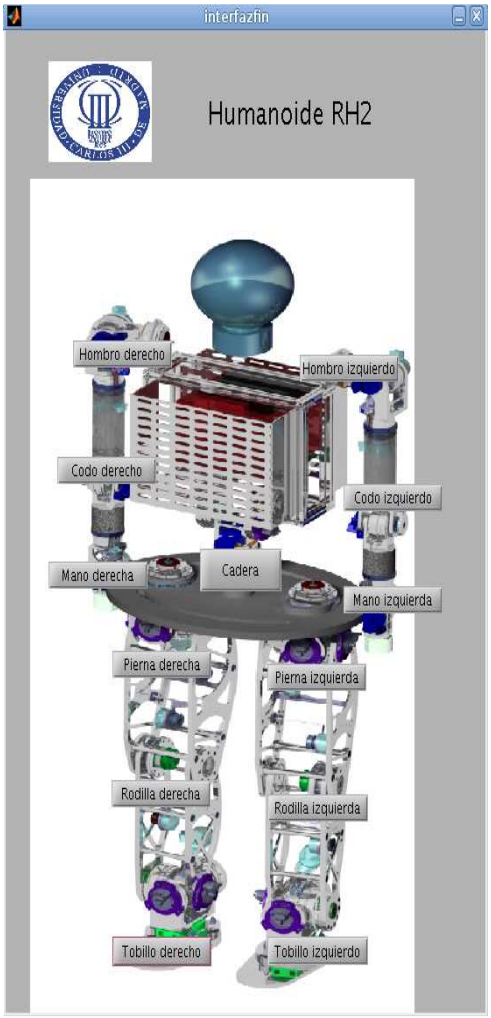


Figura 51: Interfaz final

Pulsando en cada opción, está te envía a las diferentes articulaciones para que se pueda realizar los diferentes movimientos.

La tabla 15 explica los movimientos que puede realizar el robot en cada una de sus articulaciones.

Tabla 15: Planos de cada articulación

ARTICULACIÓN	MOVIMIENTO
Tobillo	Sagital
	Frontal
Rodilla	Sagital
Pierna	Sagital
	Frontal
	Transversal
Cadera	Sagital
	Transversal
Hombro	Sagital
	Frontal
Codo	Sagital
	Transversal
Mano	Transversal

Estos movimientos son los representados en la figura 52:

- Plano sagital: divide el cuerpo en mitad derecha e izquierda.
- Plano frontal: divide el cuerpo en mitad anterior y posterior.
- Plano transverso: divide el cuerpo en mitad superior e inferior.

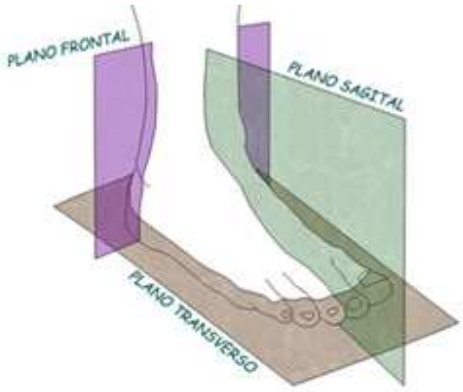


Figura 52: Planos

Una interfaz más específica es la explicada en la figura 53 que muestra el tobillo derecho, esta interfaz tiene diferentes opciones.

- Elecciones del usuario:

1. Movimiento: sagital o frontal. Cuando se realiza un cambio se llama a la función matlanodo.c para identificar el nuevo driver

```
function uipanel2_SelectionChangeFcn(hObject, eventdata, handles)
if (hObject == handles.movsagital)
   set(handles.text5, 'String', '1');
else
   set(handles.text5, 'String', '2');
end
nodo = str2double(get(handles.text5, 'String'));
matlanodo(nodo)
```

2. Velocidad: escribirá la velocidad deseada en rpm.

```
function velocidad_Callback(hObject, eventdata, handles)
velocidad= str2double(get(handles.velocidad, 'String'));
if isnan(velocidad)
   set(hObject, 'String', 0);
errordlg('Introduce un numero','Error');
end
```

3. Posición: escribirá la posición deseada en grados, por seguridad está limitado de -20 a +20º.

```
function posicion_Callback(hObject, eventdata, handles)
posicion= str2double(get(handles.posicion, 'String'));
if isnan(posicion)
   set(hObject, 'String', 0);
errordlg('Introduce un numero','Error');
end
if posicion>20
   set(hObject, 'String', 0);
errordlg('Introduce un angulo menor de 20 grados','Error');
end
if posicion<-20
   set(hObject, 'String', 0);
errordlg('Introduce un angulo mayor de -20 grados','Error');
end
```

4. Posicionar: mandara la información recogida para la realización del movimiento.

```
function posicionar_Callback(hObject, eventdata, handles)
vel = str2double(get(handles.velocidad, 'String'));
pos = str2double(get(handles.posicion, 'String'));
matlaenlace(vel,pos)
```

5. Ver: muestra la posición y velocidad de ese momento.

```
function vertiempo_gui(fig_handle, handles, isreset)
[vel,pos]=matlaver;
set(handles.text3, 'String', num2str(vel));
set(handles.text4, 'String', num2str(pos));
pause(0.5);
```

- Otras opciones:

6. Nodo: es el número del driver que contrala ese movimiento en esa articulación (van del 1 al 24) en el caso del tobillo derecho es el 1 (frontal) o el 2 (sagital).

7. Actual: es la posición y velocidad el momento.

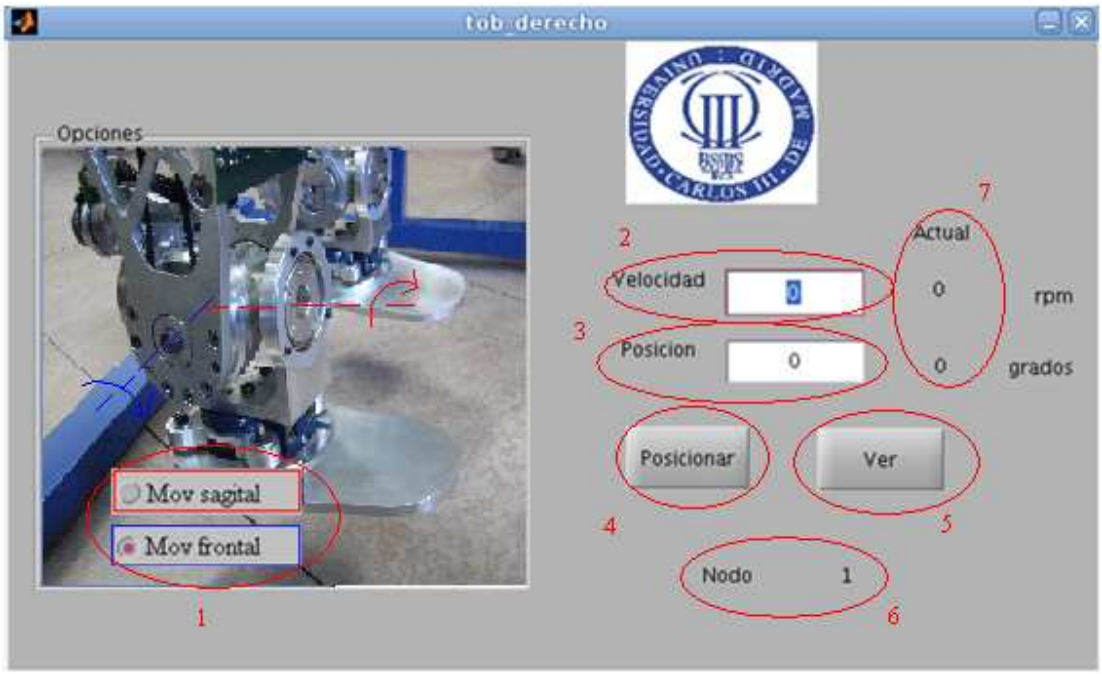


Figura 53: Interfaz tobillo derecho

Las demás articulaciones son iguales, solo cambiaran los diferentes movimientos que puedan realizar como ya se ha explicado en la tabla 14.

Para introducir los dibujos correspondientes solo es necesario realizar este programa:

```
function tob_derecho_OpeningFcn(hObject, eventdata, handles, varargin)

tobillo=imread('tobillo_derecho.jpg');
axes(handles.tobillo);
image(tobillo);
axis off

logo =imread('logo2.jpg');
axes(handles.logo);
image(logo);
axis off;
```

Donde tobillo y logo son los objetos Axes, tobillo_derecho.jpg es la imagen del tobillo y logo2.jpg es el logotipo de la universidad.

CAPITULO 5: PUESTA EN MARCHA DE LA APLICACIÓN

En este capítulo se van a observar los resultados obtenidos durante los ensayos. Esta acción se ha llevado a cabo a través de una lectura de datos usando la función xlswrite, esta función te crea un documento Excel al cual se envían los datos para realizar las gráficas.

5.1 Test 1: Movimiento a +20º eje frontal

Se ha realizado un movimiento a 20 grados y luego vuelta a 0 a diferentes velocidades para comprobar la captura de datos.

La figura 54 muestra el movimiento a una velocidad de 1rpm.

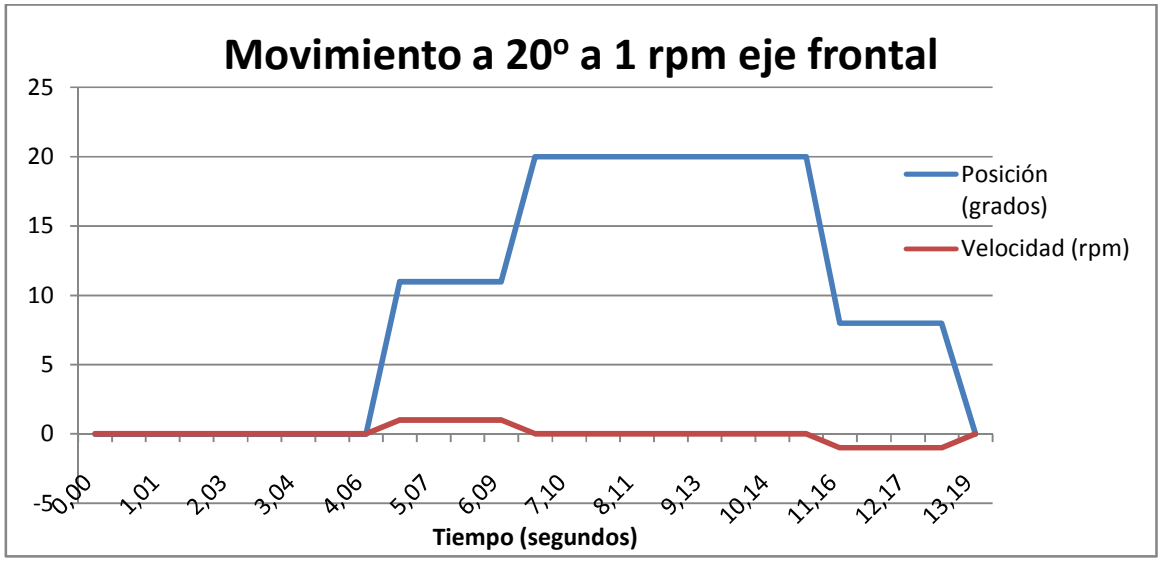


Figura 54: Movimiento a 20º a 1 rpm eje frontal

Debido a que el programa tarda en capturar los datos se produce el escalón que se observa en la figura anterior. El tiempo que tarda son 2.5 segundos en realizar la subida y otros 2.5 en la bajada.

Su grafica de intensidad dinámica es la mostrada en la figura 55 teniendo en cuenta las velocidades halladas en la figura 54.

Para el cálculo de la intensidad se usa la ecuación 5.

$$I = \frac{v}{\mathrm{Rv} * \mathrm{Cp}} \quad (5)$$

Dónde:

- I= intensidad.
- v= velocidad salida.
- Rv= relación de velocidad para (7.86rpm/mNm).
- Cp= constante de par (75.8mNm/A).

Hay que tener en cuenta la relación de transformación, los valores de velocidad dados en las tablas son los de salida, por tanto la velocidad de entrada es multiplicada por la relación de transformación, 320.

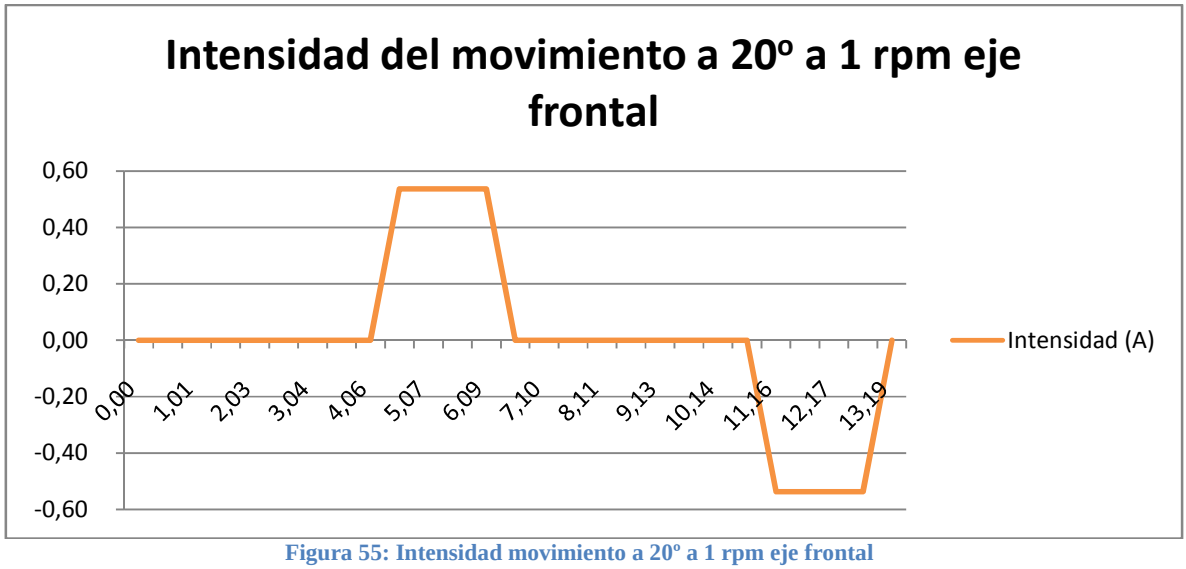


Figura 55: Intensidad movimiento a 20º a 1 rpm eje frontal

Solo se produce un pico de intensidad cuando hay una velocidad a la que asociar la formula.

La figura 56 muestra el movimiento a una velocidad de 2 rpm.

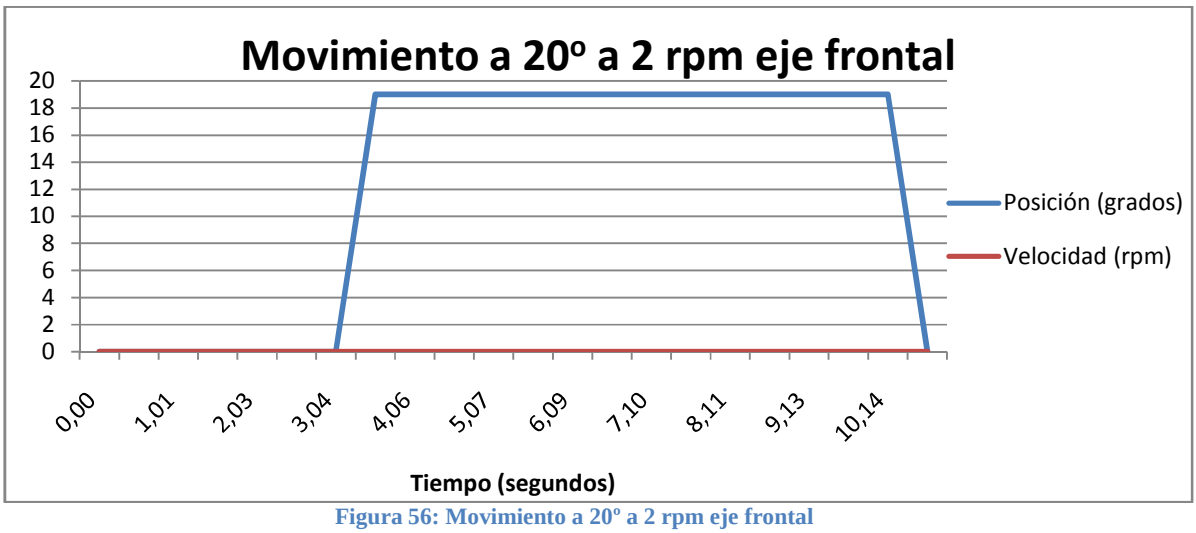


Figura 56: Movimiento a 20º a 2 rpm eje frontal

A partir de esta velocidad y esta distancia (20 grados) el programa no es capaz de capturar la velocidad.

El cálculo de la intensidad dinámica seria 0 ya que no hay ninguna velocidad.

5.2 Test2: Movimiento de +20º a -20º en el eje frontal

Se ha realizado un movimiento a 20 a -20grados (todo ello teniendo en cuenta que el enconder está en modo absoluto) a diferentes velocidades para comprobar la captura de datos.

La figura 57 muestra el movimiento a una velocidad de 1 rpm.

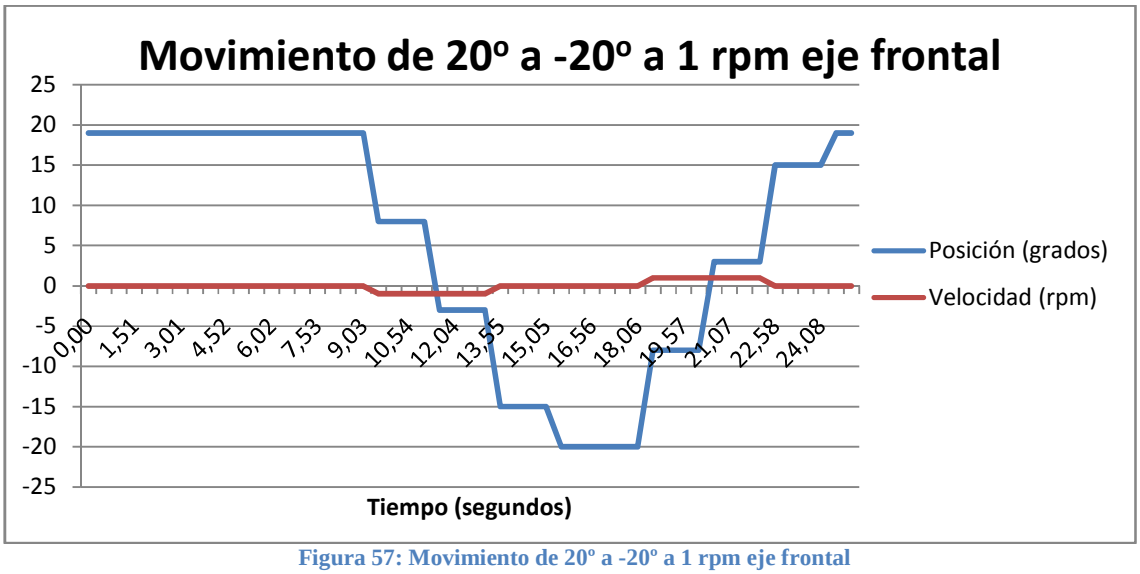


Figura 57: Movimiento de 20º a -20º a 1 rpm eje frontal

Debido a que el programa tarda en capturar los datos se produce el escalón que se observa en la figura anterior. Se puede observar cuando está en la posición -15 grados, la velocidad es 0 debido a que el programa ha podido capturar la posición pero cuando ha ido a capturar la velocidad el movimiento había terminado.

Su grafica de intensidad dinámica es la mostrada en la figura 58.

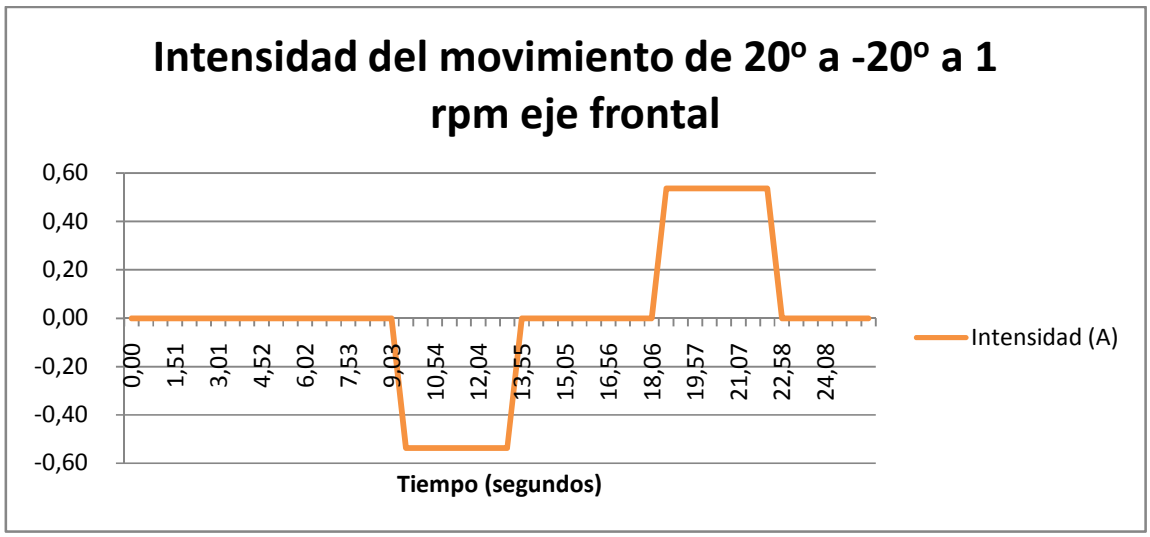


Figura 58: Intensidad movimiento de 20º a -20º a 1 rpm eje frontal

Se observa la misma intensidad que en la figura 58 que la que se produjo en la figura 55, debido a que el movimiento de velocidad es el mismo, la diferencia es la duración.

La figura 59 muestra el movimiento a una velocidad de 2 rpm.

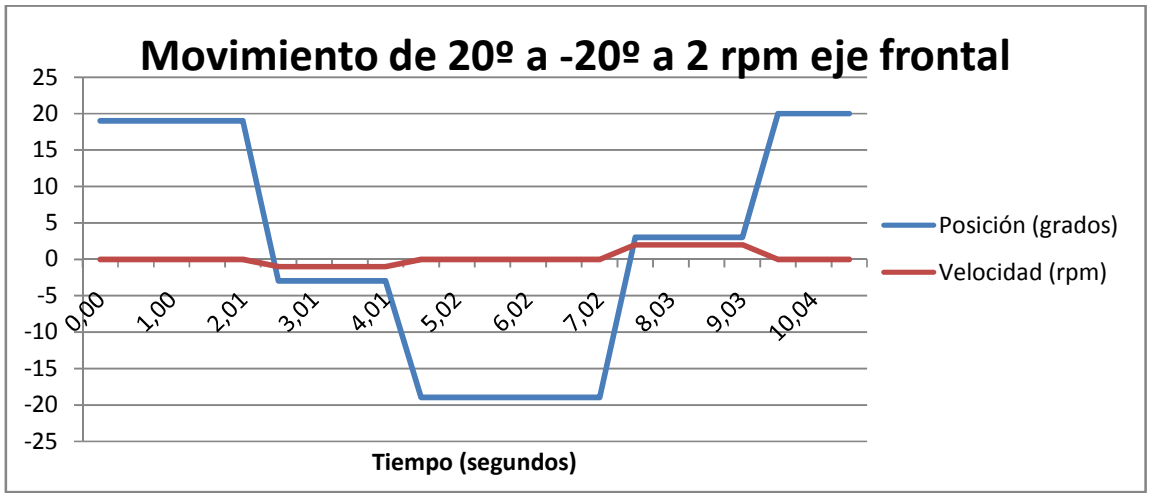


Figura 59: Movimiento de 20º a -20º a 2 rpm eje frontal

Como se puede observar hay una disminución del número de escalones debido a que cada vez el movimiento se realiza a mayor velocidad y el programa no puede seguirlo. Se puede observar también que para la primera parte del movimiento la velocidad es 1 en vez de 2, esto es un fallo de lectura al ser unidades tan pequeñas.

La gráfica de intensidad dinámica se muestra en la figura 60.

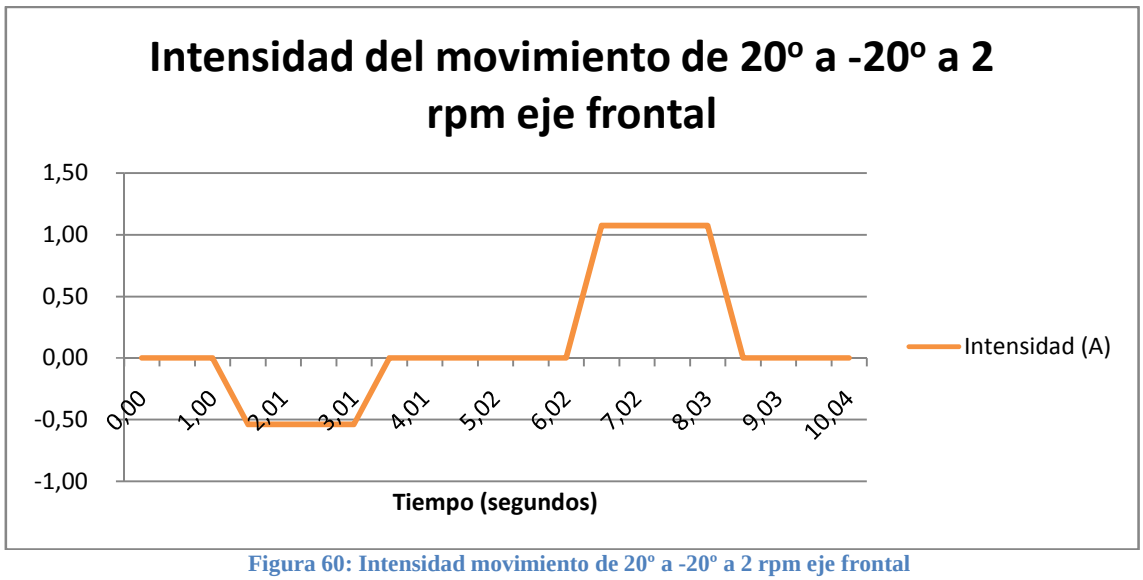


Figura 60: Intensidad movimiento de 20º a -20º a 2 rpm eje frontal

La intensidad es proporcional, al aumentar el doble la velocidad la intensidad también lo hace.

La figura 61 muestra el movimiento a una velocidad de 3 rpm.

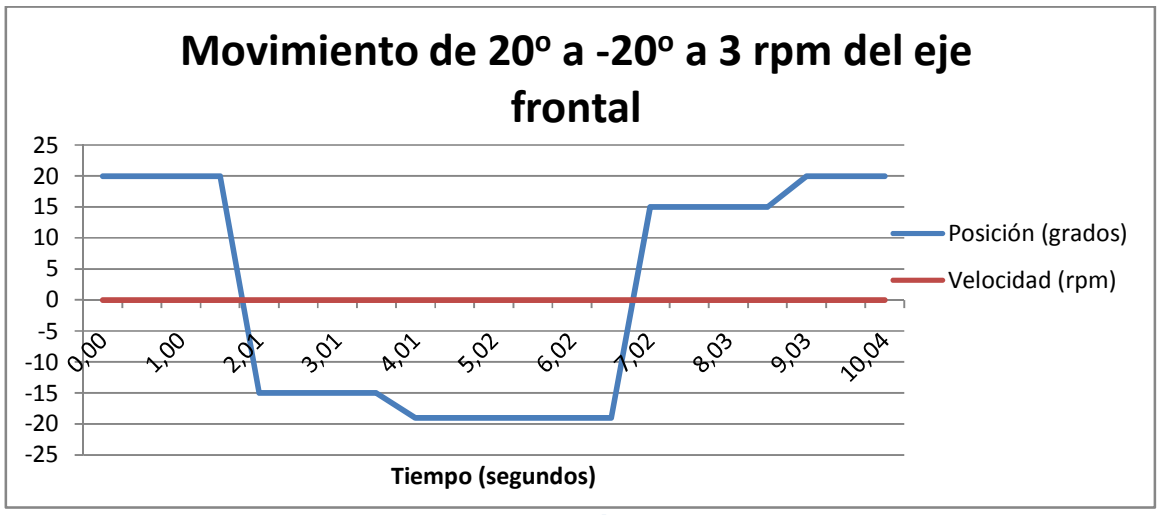


Figura 61: Movimiento de 20º a -20º a 3 rpm

En la figura anterior se puede observar como la velocidad ya es demasiado excesiva para que pueda capturar correctamente los datos, a capturado un dato de posición (que el programa se realiza en primer lugar) pero cuando ha llegado para capturar la velocidad el movimiento había finalizado. En esta figura se puede observar bien como el programa tarda entre 1.5 y 2 segundos en volver a generar un valor.

Su gráfica de intensidad dinámica es inexistente debido a la ausencia de velocidad.

5.3 Test 3: Movimiento a +20º eje sagital

En los test que vienen a continuación se van a realizar los mismos movimientos exceptuando que hay un cambio en el eje.

Se ha realizado un movimiento a 20 grados y luego vuelta a 0 a diferentes velocidades para comprobar la captura de datos.

La figura 62 muestra el movimiento a una velocidad de 1 rpm.

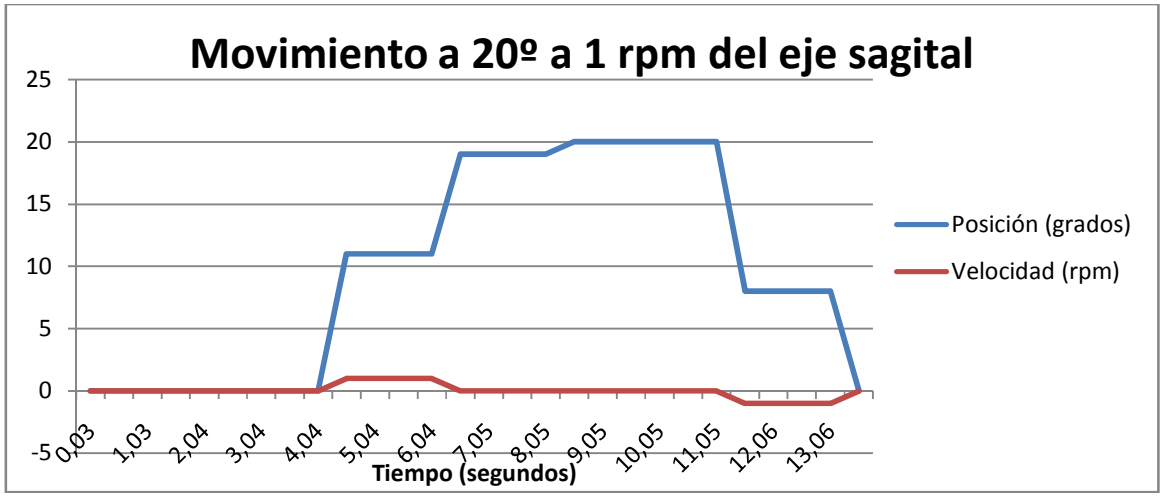


Figura 62: Movimiento a 20º a 1 rpm

Los resultados obtenidos son idénticos al anterior eje con la excepción de que antes de llegar al final del movimiento ha capturado otro valor (mientras que en la figura 54 solo se observaba un escalón).

Su gráfica de intensidad es la mostrada en la figura 63.

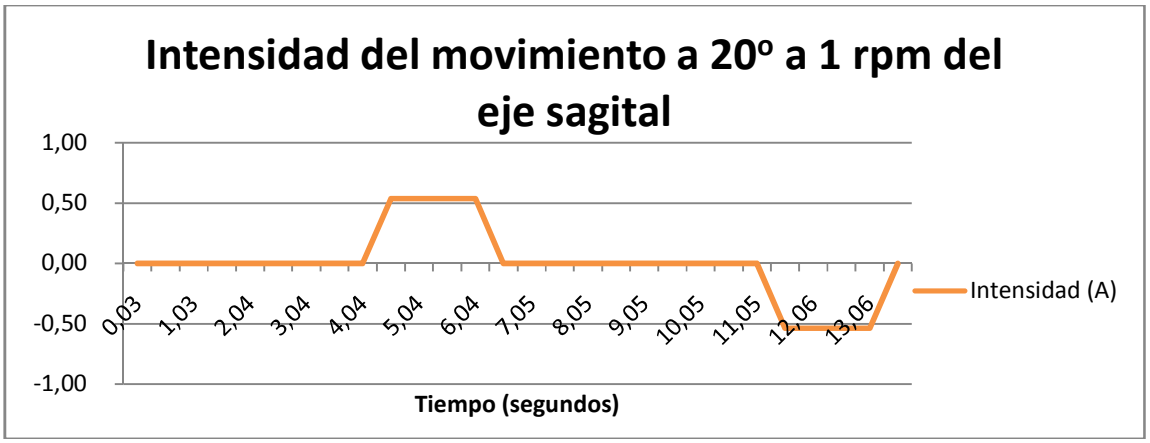


Figura 63: Intensidad movimiento a 20º a 1 rpm

La intensidad dinámica es idéntica a la figura 55.

La figura 64 muestra el movimiento a una velocidad de 2 rpm.

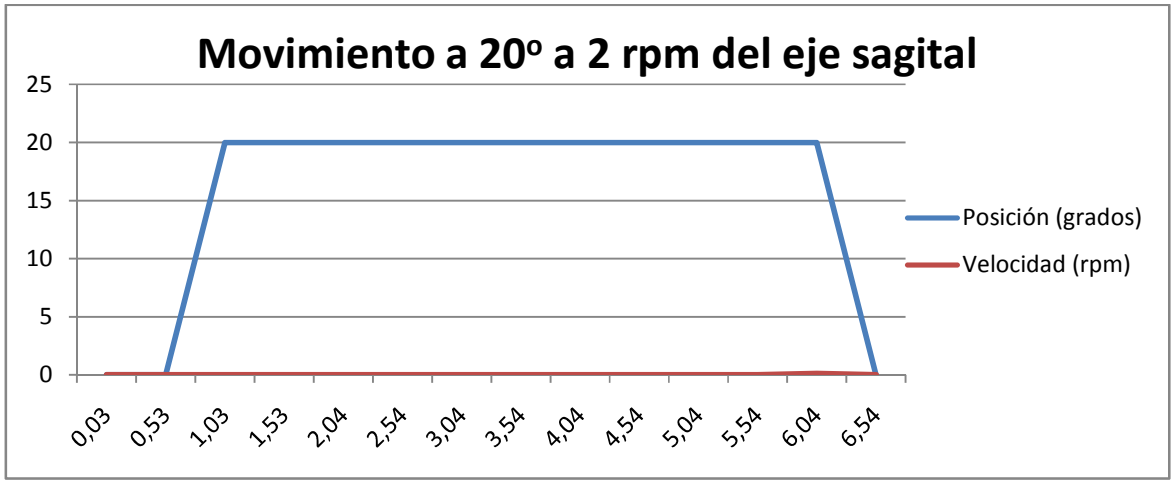


Figura 64: Movimiento a 20º a 2 rpm

Como pasa con el eje frontal a partir de 1 ya no es capaz de capturar datos de velocidad. Su intensidad dinámica es cero, ya que no existe velocidad en la gráfica con la que poder calcularla.

5.4 Test4: Movimiento de +20º a -20º en el eje sagital

Se ha realizado un movimiento a 20 a -20grados (todo ello teniendo en cuenta que el enconder está en modo absoluto) a diferentes velocidades para comprobar la captura de datos:

La figura 65 muestra el movimiento a una velocidad de 1 rpm.

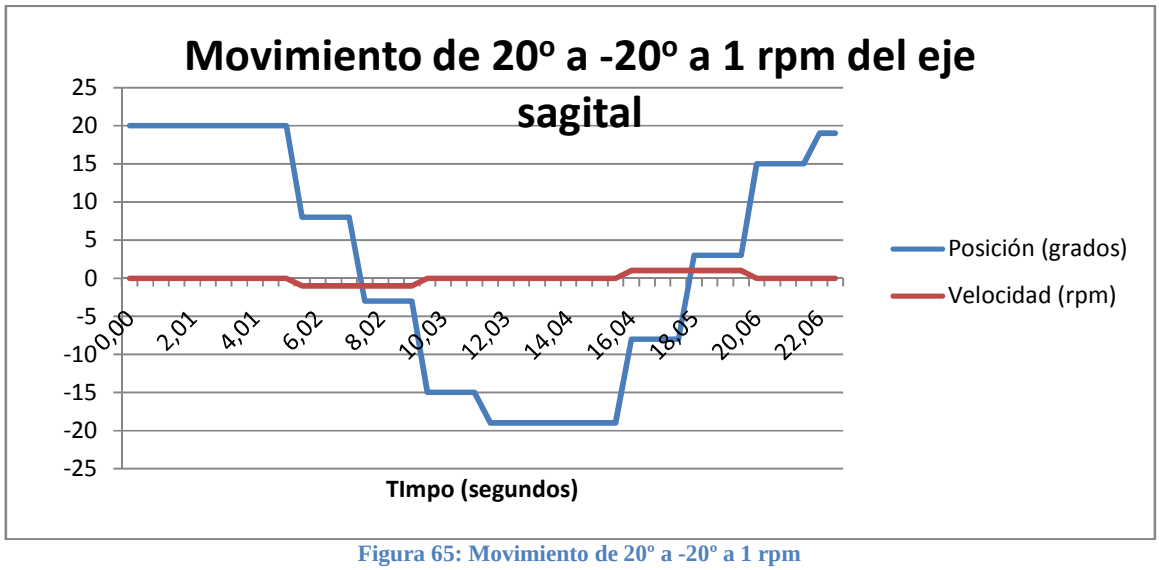


Figura 65: Movimiento de 20º a -20º a 1 rpm

Se puede observar lo mismo que en la figura 57, los escalones que se producen debido al retardo en actualizar los datos.

Su gráfica de intensidad dinámica es la mostrada en la figura 66.

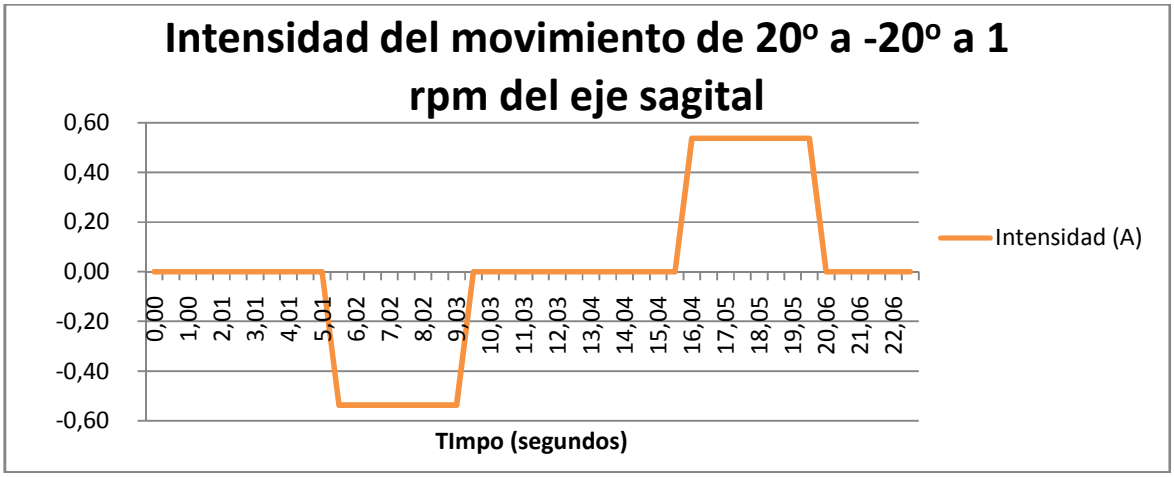


Figura 66: Intensidad movimiento de 20º a -20º a 1 rpm

Como está pasando en todas las gráficas, son idénticas a las calculadas para el otro eje

La figura 67 muestra el movimiento a una velocidad de 2 rpm.

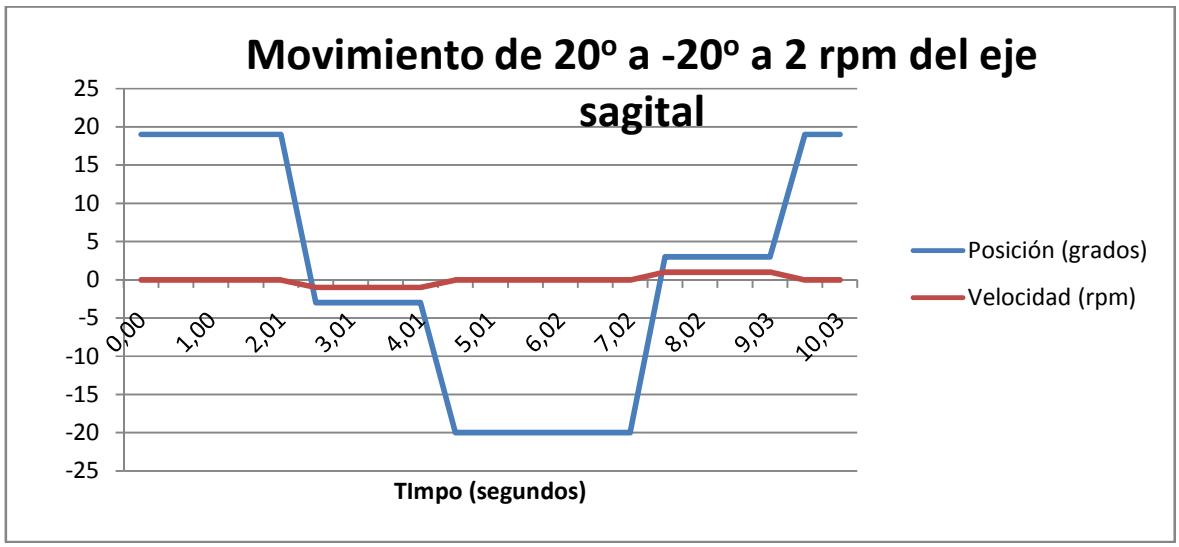


Figura 67: Movimiento de 20º a -20º a 2 rpm

Aunque se haya puesto en la interfaz 2 rpm, el programa cuando lee los datos recibidos por el driver sigue diciendo que es 1 rpm.
El retardo que tarda el programa en tomar un nuevo valor es de 1.5 segundos, como se observa perfectamente en la figura 67.

La gráfica de intensidad asociada se muestra en la figura 68.

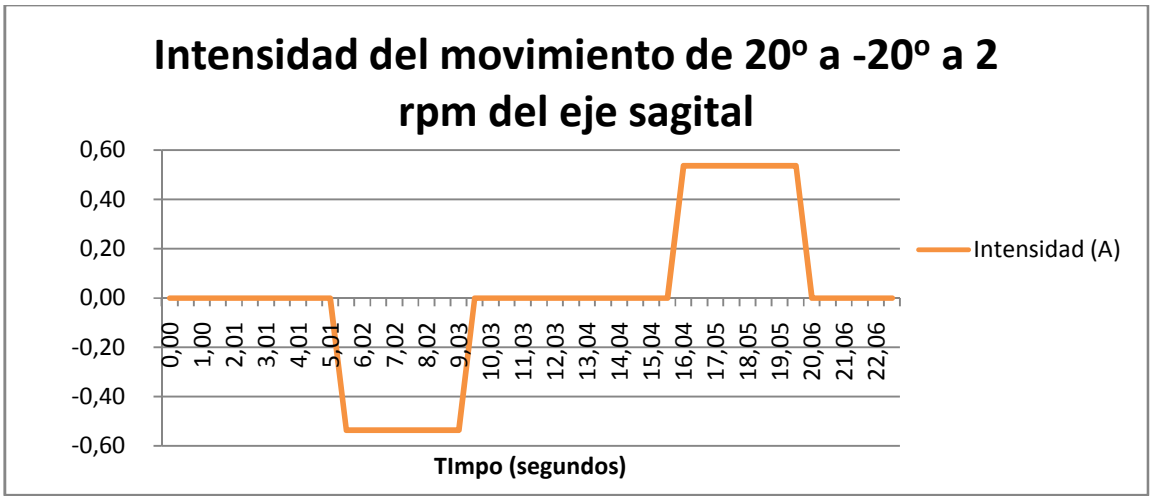


Figura 68: Intensidad movimiento de 20º a -20º a 2 rpm

Ya que en la figura 67 la velocidad es 1, aunque teóricamente debería ser 2, su gráfica de intensidad que se muestra en la figura 68 es igual que para 1 rpm.

La figura 69 muestra el movimiento a una velocidad de 3 rpm.

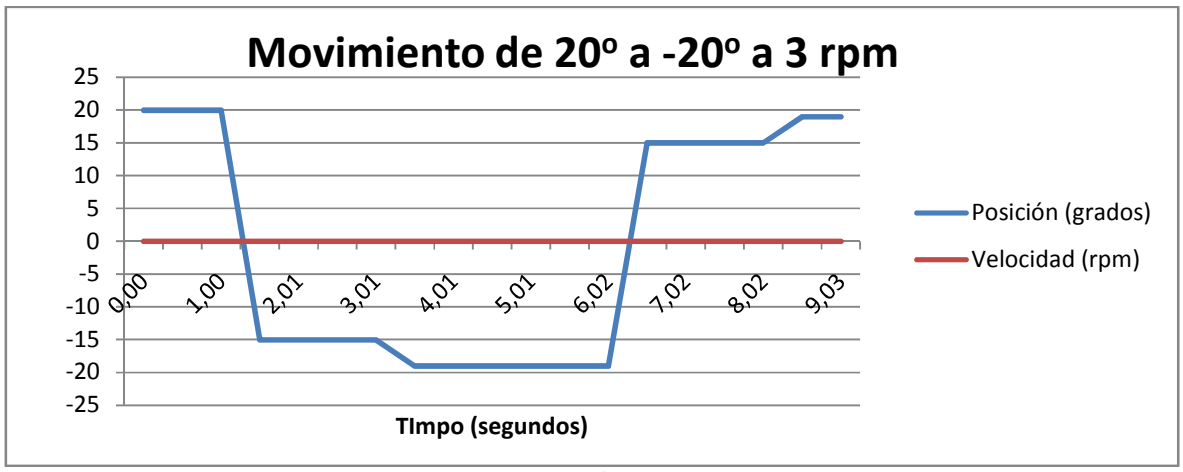


Figura 69: Movimiento de 20º a -20º a 3 rpm

Como ha pasado para el anterior eje, para velocidades superiores a 2 rpm el programa no es capaz de capturar la velocidad.

CAPITULO 6: CONCLUSIONES

6.1 Conclusiones

En los ensayos se ha demostrado que para una distancia de 20 grados la velocidad no pude ser superior a 1 rpm si se quiere observar que hay un cambio de velocidad, por tanto como también se ha observado para un cambio de 40 grados la velocidad no puede ser superior a 2 rpm y así proporcionalmente.

También se ha observado que no hay diferencia entre el eje sagital y el frontal en cuestiones de movimiento.

Se han cumplidos los objetivos que fueron planteados para este proyecto:

- Se ha implementado un control directo que facilitará el desarrollo de futuros algoritmos de control para el robot RH-2 bajo la plataforma de lenguaje C

- Se ha desarrollado un software que realiza el control directo y la monitorización de los parámetros de los motores que controlan las articulaciones del robot humanoide mediante el protocolo de comunicación CANOpen.

- Se ha desarrollado mediante la herramienta de Matlab una interfaz gráfica para el uso del usuario.

- Se han realizado los ensayos y demostraciones oportunas para la comprobación del correcto funcionamiento del software desarrollado y del hardware empleado.

Dado que el RH-2 se encuentra aún en fase de desarrollo, todas las pruebas han sido realizadas sobre un péndulo invertido de dos grados de libertad con una estructura semejante al tobillo del robot.

El robot final usa unos motores brushless en vez de brushed, el único cambio para solucionar eso es introducir en la configuración del motor (4.2.2) los nuevos datos del nuevo motor.

En cambio, los drivers ISCM8005 de TechnoSoft si van a ser utilizados en el robot RH-2. Hay que tener en cuenta si quiere cambiar el modo de posición a velocidad usar el programa EasySetUp.

Durante la realización de este proyecto se han encontrado diferentes problemas para poder desarrollarlo:

- El más importante fue, al principio del proyecto, como enviar los datos por medio de CAN. No era solo como enviarlo sino también como poder guardarlos en una variable para poder enviarlos, después de muchos intentos se decide hacerlo por la forma que está ahora: mensaje.campo_donde_se_quiere_guardar = valor (un ejemplo es mensaje.dlc=2). Ya para enviar el mensaje se realiza la siguiente instrucción, que si está explicada con más detalle en el manual: write(nodoTx, &mensaje, sizeof (struct can_msg));

- El manual de CAN si te explica muy bien como abrir un nodo pero no te dice lo mismo con los pasos para inicializarlo, que se explicaron en el capítulo 3.2.9. Esos números (6,7,F) que hay que introducir fueron bastante difícil de encontrar.

- Un problema muy sencillo de arreglar pero que puedes tener muchos problemas si no te das cuenta es la velocidad de comunicación (baudios), el driver y el programa C tienen que tener la misma velocidad, en caso de que no fuese así, no funciona nada.

- El último problema fue la comunicación entre la interfaz y el programa C, al final se decide hacer por memorias compartidas después de intentarlo a través de colas y tubería.

6.2 Trabajos futuros

A pesar del trabajo realizado se pueden realizar ciertas mejoras que mejorarían las prestaciones del programa. Un ejemplo de ellas serían las siguientes:

- El principal trabajo futuro sería acortar los tiempo de respuesta para que se pueda mostrar los valores reales, es decir, ahora mismo el programa tarda 1.5 segundos en refrescar un nuevo valor, ese tiempo es excesivo si se quiere realizar un control a tiempo real.

- Conseguir movimientos fluidos para evitar los parones, debido a que se ha observado que cuando se aumenta la velocidad de movimiento se produce un pequeño golpe al llegar la articulación a su destino, ya que el motor no ajusta su velocidad cuando va a llegar al final del recorrido y se para bruscamente.

- Realizar una interfaz más compleja para observar los movimientos realizados durante un periodo de tiempo, ahora mismo los datos recibidos se mandan a un archivo Excel donde se analizan y se realiza las gráficas oportunas, esta opción pude estar dentro de la propia interfaz en un botón que al presionarlo saque los últimos movimientos realizados en una gráfica.

- Poder guardar las maniobras que realice el usuario si este quiere para su continua repetición, es decir, si hay una maniobra típica como por ejemplo la caminata, la interfaz estará dotada de unas opciones de grabación para poder guardar las indicaciones obtenidas la primera vez y así poder repetirlas en próximas sucesiones.

- Conseguir cambiar el movimiento de la articulación antes de que llegue al destino fijado, si se produce un cambio en la elección del usuario. Esto también serviría para evitar accidentes.

CAPITULO 7: PRESUPUESTO

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

PRESUPUESTO DE PROYECTO

1.- Autor: Raúl Morales Tejero

2.- Departamento: Ingeniería de Sistemas y Automática

3.- Descripción del Proyecto
- Titulo: desarrollo de una aplicación para el control directo de las articulaciones del robot humanoide RH-2
- Duración (meses):15 meses
- Tasa de costes

Indirectos: aplicando un 15% 2250€

4.- Presupuesto total del Proyecto

5000 Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a título informativo)	Categoría	Dedicación (hombres mes) a)	Coste hombre mes	Coste (Euro)	Firma de conformidad
Raúl Morales			30	1000	15000	
				Total	15000	

a) 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas) Máximo anual para PDI de la Universidad Carlos III de Madrid

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d)]
Ordenador	600	100	15	60	15000
DRIVER ISCM8005	800	100	10	60	13333
TARJETA CAN	350	100	10	60	5833
				Total	34166

[d)] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	15000
Amortización	0
Subcontratación de tareas	0
Costes de funcionamiento	34166
Costes Indirectos	2250
Total	51416

BIBLIOGRAFIA

Documentos

- [1] ISCM4805ISCM8005*Version 1.3.* Intelligent Servo Drive for Step, DC, Brushless DC and AC Motors. Technosoft
- [2]QNX 6.3 CAN driver. HiCO.CAN MiniPCI, PCI-104 and PC/104+ boards
- [3]CANOpen high-level protocol for CAN-bus. H. Boterenbrood. NIKHEF, Amsterdam March 20, 2000
- [4]CANOpen Programming. Technosoft
- [5]ISCMxx05 SK-ST V1.2 Starter Kit for ISCMxx05 Intelligent Stepper Drive – Step Motor. Technosoft
- [6]Linux CAN driver manual. HiCO.CAN MiniPCI, PCI-104 and PC/104+ boards
- [7]Driver Guide HiCO.CAN PCI/ISA driver for Linux

Páginas de internet

- [8]http://export.rsdelivers.com/product/maxon/251601/ec-45-50w-flat-motor/0495063.aspx Accedido en marzo de 2011
- [9]http://canbus.galeon.com/electronica/canbus.htm Accedido en abril de 2010
- [10]http://es.wikipedia.org/wiki/Bus_CAN Accedido en mayo de 2010
- [11]http://catarina.udlap.mx/u_dl_a/tales/documentos/lep/garcia_b_s/capitulo3.pdf Accedido en diciembre de 2010

Proyectos fin de carrera

- Alberto Navarro Criado, “Análisis de los accionadores del robot humanoide RH-2”, Universidad Carlos III de Madrid. Fecha de lectura 2009.
- Guillermo Gallardo Fernández, “Caracterización y control de los motores del tobillo del robot humanoide RH-2”, Universidad Carlos III de Madrid. Fecha de lectura 2010.

- Julián Fortes Monteiro, “Diseño e implementación encoder absoluto CANOpen”, Universidad Carlos III de Madrid. Fecha de lectura 2009.

- Andrés Cano Sánchez, “Diseño de la arquitectura hardware del robot humanoide RH-2”, Universidad Carlos III de Madrid. Fecha de lectura 2008.

Tesis fin de máster

- José Álvarez Paramio, “Diseño e implementación de una arquitectura para el control del robot humanoide RH-2”, Universidad Carlos III de Madrid. Fecha de lectura 2011.

ANEXOS

Motor brushed 273757 de Maxon

EC 45 flat Ø45 mm, brushless, 50 Watt

maxon flat motor

Stock program
Standard program
Special program (on request)

		Order Number			
	with hall sensors	339285	231001	339286	339287
Motor Data					
	Values at nominal voltage				
1	Nominal voltage	V	18.0	24.0	24.0	36.0
2	No load speed	rpm	6710	6700	4700	3360
3	No load current	mA	294	201	116	46.9
4	Nominal speed	rpm	5250	5260	3410	2320
5	Nominal torque (max. continuous torque)	mNm	96.9	94.3	94.2	94.3
6	Nominal current (max. continuous current)	A	3.54	2.36	1.32	0.951
7	Stall torque	mNm	1100	822	397	469
8	Starting current	A	43.6	24.5	7.10	4.86
9	Max. efficiency	%	84.5	83	77	82
	Characteristics					
10	Terminal resistance phase to phase	Ω	0.413	0.978	3.38	7.41
11	Terminal inductance phase to phase	mH	0.322	0.573	1.15	5.15
12	Torque constant	mNm / A	25.1	33.5	47.5	101
13	Speed constant	rpm / V	380	285	201	95.0
14	Speed / torque gradient	rpm / mNm	6.84	8.32	14.3	7.00
15	Mechanical time constant	ms	9.82	11.8	20.3	9.90
16	Rotor inertia	gcm²	135	135	135	135

Specifications

Thermal data
17 Thermal resistance housing-ambient 4.25 K / W
18 Thermal resistance winding-housing 4.5 K / W
19 Thermal time constant winding 16.8 s
20 Thermal time constant motor 212 s
21 Ambient temperature -40 ... +100°C
22 Max. permissible winding temperature +125°C

Mechanical data (preloaded ball bearings)
23 Max. permissible speed 10000 rpm
24 Axial play at axial load < 4.0 N 0 mm
> 4.0 N 0.14 mm
25 Radial play preloaded
26 Max. axial load (dynamic) 3.8 N
27 Max. force for press fits (static) 50 N
(static, shaft supported) 1000 N
28 Max. radial loading, 7.5 mm from flange 5.5 N

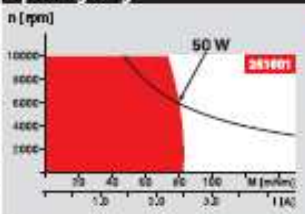
Other specifications
29 Number of pole pairs 9
30 Number of phases 3
31 Weight of motor 110 g

Values listed in the table are nominal.

Connection
Pin 1 Hall sensor 1*
Pin 2 Hall sensor 2*
Pin 3 4.5 ... 18 VDC
Pin 4 Motor winding 3
Pin 5 Hall sensor 3*
Pin 6 GND
Pin 7 Motor winding 1
Pin 8 Motor winding 2
*Internal pull-up (7 ... 13 kΩ) on pin 3
Wiring diagram for Hall sensors see page 29

Cable
Connection cable Universal, L = 500 mm 339380
Connection cable to EPOS, L = 500 mm 354046

Operating Range



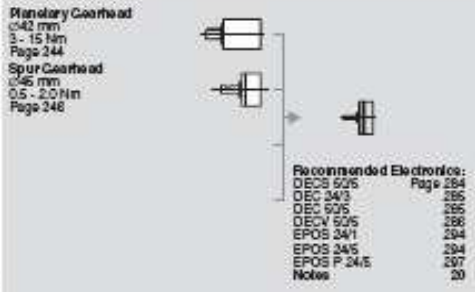
Comments

Continuous operation
In observation of above listed thermal resistance (lines 17 and 18) the maximum permissible winding temperature will be reached during continuous operation at 25°C ambient.
= Thermal limit.

Short term operation
The motor may be brisly overloaded (recurring).

Assigned power rating

maxon Modular System Overview on page 18 - 21



Planetary Gearhead
Ø42 mm
3 - 15 Nm
Page 244

Spur Gearhead
Ø45 mm
0.5 - 2.0 Nm
Page 248

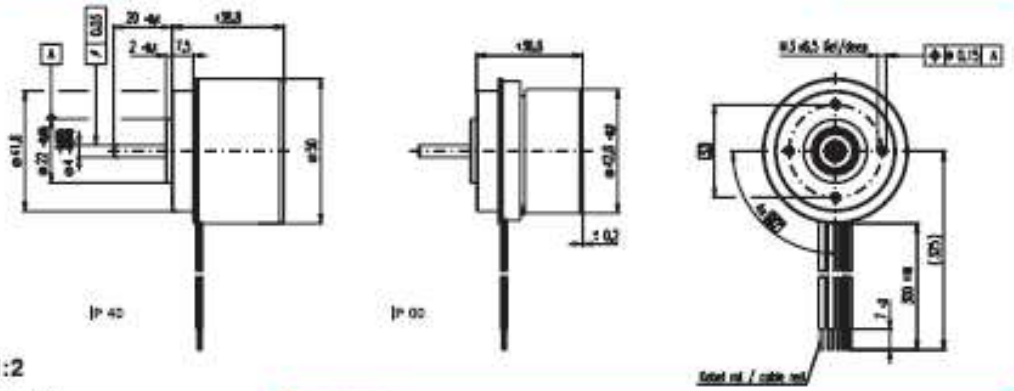
Recommended Electronics:

DECS 50/5	Page 284
DEC 24/3	285
DEC 50/5	285
DECV 50/5	286
EPOS 24/1	294
EPOS 24/5	294
EPOS P 24/5	297
Notes	20

EC 45 flat brushless, 50 Watt, with integrated electronics

NEW

1-Q-speed controller



Stock program
Standard program
Special program (on request)

Order Number	2 wire version		5 wire version	
IP 40 (with cover)	353026		352810	
IP 00 (without cover)		353024		352887

Motor Data (provisional)						
	Values at nominal voltage					
1	Nominal voltage	V	24	24	24	24
2	No load speed	rpm	3000	3000	4500	4500
3	No load current	mA	127	127	192	192
4	Nominal speed	rpm	3000	3000	4500	4500
5	Nominal torque (max. continuous torque)	mNm	66	123	71	118
6	Nominal current (max. continuous current)	A	1.2	2.4	2	3
33	Max. torque	mNm	154	154	154	154
34	Max. current	A	3.3	3.3	3.8	3.8
3	Max. efficiency	%	74	74	78	78
	Characteristics					
35	Control variable		Speed	Speed	Speed	Speed
36	Supply voltage + V_{cc}	V	10 .. 28	10 .. 28	10 .. 28	10 .. 28
37	Speed set value input	V	= V_{cc}	= V_{cc}	0,35 .. 10,8	0,35 .. 10,8
38	Scale speed set value input	rpm / V	125	125	600	600
39	Turning speed range	rpm	1250 ... 3500	1250 ... 3500	200 .. 6480	200 ... 6480
40	Max. acceleration	rpm / s	3000	3000	6000	6000

Specifications

	Thermal data	
17	Thermal resistance housing-ambient	5.8 (2.4) K / W
18	Thermal resistance winding-housing	4 K / W
19	Thermal time constant winding	27 s
20	Thermal time constant motor	280 (116) s
21	Ambient temperature	-40 .. +85°C
22	Max. permissible winding temperature	+125°C
41	Max. temperature of electronics	+105°C
	Mechanical data (preloaded ball bearings)	
16	Rotor inertia	183 gcm²
24	Axial play at axial load < 2 N	0 mm
	> 2 N	0.14 mm
25	Radial play	preloaded
26	Max. axial load (dynamic)	6.4 N
27	Max. axial load (static)	85 N
	(static, shaft supported)	1000 N
28	Max. radial loading, 5 mm from flange	34 N
	Other specifications	
31	Weight of motor	280 g
32	Direction of rotation	Clockwise (CW)

Values listed in the table are nominal.

Protective functions
Overload protection, blockage protection, inverse-polarity protection, thermal overload protection, low/high voltage cutoff

Connection 2 wire version (Cable AWG 18/24)

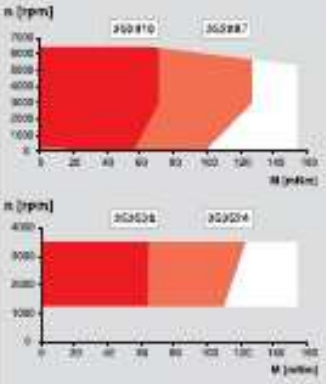
red	+ V_{cc} 10 .. 28 VDC
black	GND

Connection 5 wire version (Cable AWG 18/24)

red	+ V_{cc} 10 .. 28 VDC
black	GND
white	Speed set value input
green	Monitor n (8 pulses per revolution)
grey	Disable (2,4 .. 28 VDC)

May 2006 edition / subject to change

Operating Range



Comments

Continuous operation
The drive can be operated with a speed controller and, taking account of the given thermal resistance (fig. 17 and 18) at an ambient temperature of 25°C, does not exceed the maximum permissible operating temperatures.

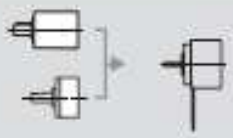
Overload range
The drive reaches these operating points. Speed may vary from the set value. The overload protection shuts down the drive in the event of sustained overload.

maxon Modular System

Overview on page 16 - 21

Planetary Gearhead
Ø42 mm
3 - 15 Nm
Page 244

Spur Gearhead
Ø45 mm
0.5 - 2.0 Nm
Page 246

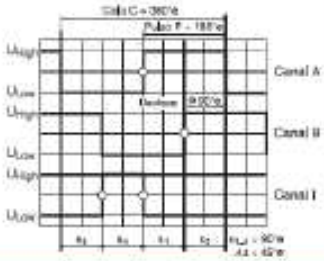


maxon flat motor

Encoder relativo 228452 de Maxon

Encoder MR, tipo L, 256 - 1024 ppv, 3 canales, con line driver

maxon sensor



Programa Stock
Programa Estándar
Programa Especial (previo encargo)

Números de Referencia	225783	228452	225785	228455	225787
Tipo					
Número de pulsos por vuelta	256	500	512	1000	1024
Número de canales	3	3	3	3	3
Máx. frecuencia de funcionamiento (kHz)	90	200	160	200	320
Máx. velocidad (rpm)	18750	24000	18750	12000	18750

Sistema Modular maxon

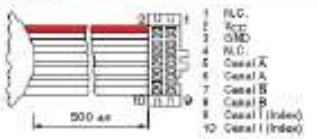
+ Motor	Página	+ Reductor	Página	+ Freno	Página	Longitud total [mm] (• ver reductor				
RE 30, 60 W	80					78.4	78.4	78.4	78.4	78.4
RE 30, 60 W	80	GP 32, 0.75 - 6.0 Nm	234/235			•	•	•	•	•
RE 30, 60 W	80	GP 32 S	251-253			•	•	•	•	•
RE 30, 60 W	80	GP 32, 0.75 - 4.5 Nm	232			•	•	•	•	•
RE 35, 90 W	81					82.4	82.4	82.4	82.4	82.4
RE 35, 90 W	81	GP 32, 0.75 - 4.5 Nm	232			•	•	•	•	•
RE 35, 90 W	81	GP 32, 0.75 - 6.0 Nm	234/235			•	•	•	•	•
RE 35, 90 W	81	GP 32, 4.0 - 8.0 Nm	237			•	•	•	•	•
RE 35, 90 W	81	GP 42, 3 - 15 Nm	240			•	•	•	•	•
RE 35, 90 W	81	GP 32 S	251-253			•	•	•	•	•
RE 40, 150 W	82					92.4	92.4	92.4	92.4	92.4
RE 40, 150 W	82	GP 42, 3 - 15 Nm	240			•	•	•	•	•
RE 40, 150 W	82	GP 52, 4 - 30 Nm	243			•	•	•	•	•
A-max 32	114/116					72.7	72.7	72.7	72.7	72.7
A-max 32	114/116	GP 32, 0.75 - 6.0 Nm	234/236			•	•	•	•	•
A-max 32	114/116	GS 38, 0.1 - 0.6 Nm	239			•	•	•	•	•
A-max 32	114/116	GP 32 S	251-253			•	•	•	•	•
EC-max 40, 70 W	172					79.9	79.9	79.9	79.9	79.9
EC-max 40, 70 W	172	GP 42, 3 - 15 Nm	241			•	•	•	•	•
EC-max 40, 120 W	173					103.9	103.9	103.9	103.9	103.9
EC-max 40, 120 W	173	GP 52, 4 - 30 Nm	244			•	•	•	•	•
ECI 40, 50 W	192					42.0	42.0	42.0	42.0	42.0
ECI 40, 50 W	192	GP 32, 1 - 6 Nm	236			•	•	•	•	•
ECI 40, 50 W	192	GP 32 S	251-253			•	•	•	•	•
ECI 40, 70 W	193					52.0	52.0	52.0	52.0	52.0
ECI 40, 70 W	193	GP 32, 1 - 6 Nm	236			•	•	•	•	•
ECI 40, 70 W	193	GP 32 S	251-253			•	•	•	•	•

Datos técnicos

Tensión de alimentación	5 V ± 5 %
Señal de salida	TTL compatible
Desfase Φ	90° ± 45°
Anchura de pulso índex	90° ± 45°
Rango de temperatura	-25 ... +85°C
Momento de la inercia de la rueda de código	< 1.7 gcm²
Corriente de salida por canal	máx. 5 mA

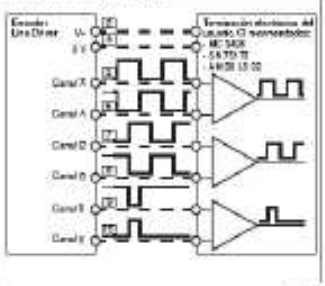
La señal del canal índex I es sincronizada con el canal A y con el B.

Conexión



Conector DIN 41651
Cable plano AWG 28

Ejemplo de conexión



Edición de octubre de 2000 / Sujeto a modificaciones

maxon sensor 265

Cara A driver ISCM8005

Pin	Nombre del pin	Tipo	Función
A1	+V mot	I	Terminal positivo de la alimentación del motor: 12 a 80V DC
A2	A / A+	O	Fase A para motores brushless Fase A+ para motores de 2 fases Fase U para motores de 3 fases Terminal positivo para motores brushed DC
A3	B / A-	O	Fase B para motores brushless Fase A- para motores de 2 fases Fase V para motores de 3 fases Terminal negativo para motores brushed DC
A4	C/B+	O	Fase C para motores brushless Fase B+ para motores de 2 fases Fase W para motores de 3 fases
A5	FRENO/B-	O	Salida freno Fase B- para motores de 2 fases
A6	+5V	O	5V generados internamente
A7	ENCA+	I	Señal A del encoder Señal A+ del encoder diferencial
A8	ENCB+	I	Señal B del encoder Señal B+ del encoder diferencial
A9	ENCZ / CAPI+	I	Señal Z del encoder Señal Z+ del encoder diferencial
A10	H1	I	Señal 1 del sensor Hall
A11	IO#38 / PULSE	I/O	Salida digital de 3.3V Entrada digital de 5V
A12	IN#2 / LSP	I	Entrada de 5V
A13	ENABLE	I	Conectar a +5V para desactivar las salidas PWM
A14	IO#13 / FDBK	I/O	Salida digital de 3.3V Entrada digital de 5V Entrada analógica de 0V a 5V. Se puede usar como realimentación analógica
A15	GND	-	Tierra
A16	CAN_H	I/O	Línea positiva del bus CAN

Cara B driver ISCM8005

Pin	Nombre del pin	Tipo	Función
B1	+V mot	I	Terminal positivo de la alimentación del motor 12 a 80V DC
B2	A / A+	O	Fase A para motores brushless Fase A+ para motores de 2 fases Fase U para motores de 3 fases Terminal positivo para motores brushed DC
B3	B / A-	O	Fase B para motores brushless Fase A- para motores de 2 fases Fase V para motores de 3 fases Terminal negativo para motores brushed DC
B4	C/B+	O	Fase C para motores brushless Fase B+ para motores de 2 fases Fase W para motores de 3 fases
B5	FRENO/B-	O	Salida freno Fase B- para motores de 2 fases
B6	+V log	O	Terminal positivo de alimentación de la lógica del driver: +12 a +48 VDC
B7	ENCA-/LH1	I	Señal A del encoder Señal A- del encoder diferencial
B8	ENCB-/LH2	I	Señal B del encoder Señal B- del encoder diferencial
B9	ENCZ- / LH3	I	Señal Z del encoder Señal Z- del encoder diferencial
B10	H2	I	Señal 2 del sensor Hall
B11	H3	I	Señal 3 del sensor Hall
B12	IN#24 / LSN	I	Entrada de 5V
B13	RESET	I	Conectar a +5V para resetear la tarjeta
B14	IO#14 / REF/DIR	I/O	Salida digital de 3.3V Entrada digital de 5V Entrada analógica de 0V a 5V. Se puede usar como realimentación de posición analógica o
B16	CAN_L	I/O	Línea negativa del bus CAN
B17	RX232	O	Línea de recepción del puerto RS-232
A17	TX232	O	Línea de transmisión del puerto RS-232

```
/*************************programa principal*****************************/
#include"cabecera.h"

int main(int argc, char *argv[]){

uint8_t buf4[8] = {0x2F,0x60,0x60,0x00,0x03,0x00,0x00,0x00};
//uint8_t buf5[8] = {0x23,0xFF,0x60,0x00,0xAB,0xAA,0xFC,0xFF}; //-100 rpm
uint8_t buf5[8] = {0x23,0xFF,0x60,0x00,0x00,0x00,0x14,0x00}; //600 rpm
uint8_t buf6[8] = {0x2F,0x60,0x60,0x00,0x01,0x00,0x00,0x00}; //modo operación
posición
uint8_t buf7[8] = {0x23,0x7A,0x60,0x00,0x40,0x1F,0x00,0x00}; //grados girar
uint8_t buf8[8] = {0x23,0x81,0x60,0x00,0xAC,0xAA,0x10,0x00}; //velocidad girar
uint8_t buf9[8] = {0x2B,0x40,0x60,0x00,0x1F,0x00,0x00,0x00};
uint8_t buf10[8] = {0x2B,0x40,0x60,0x00,0x0F,0x00,0x00,0x00};

int Vrpm,Pg,Vdriver,Pdriver;
int N=500; //líneas del encoder
int Tr=320; //relaciÃ³n de transformación
float T=0.001; //1ms
float Vui,Pui; // velocidad y posición en unidades internas
int ret, menu;
int Panterior=0;
int Nanterior;


int shmid;
memo *querida;
ver *loquehay;

pid_t pid;
/* CreaciÃ³n de la zona de memoria compartida */
if((shmid = shmget(CLAVE_SHM, sizeof(memo), IPC_CREAT|0666)) == -1){
        perror("shmget");
        exit(EXIT_FAILURE);
}

/* ObtenciÃ³n del puntero a la estructura de datos compartida */
querida = (memo *)shmat(shmid,0,0);



/*la otra zona*/
if((shmid = shmget(CLAVE_ver, sizeof(ver), IPC_CREAT|0666)) == -1){
        perror("shmget");
        exit(EXIT_FAILURE);
```

```
}

loquehay = (ver *)shmat(shmid,0,0);

inicializacion();
for (idmotor=1;idmotor<3;idmotor++)
      {  printf("modo\n");
                              mensaje.id=0x600+idmotor;
                              mensaje.dlc=8;
                        memcpy(mensaje.data,buf6,8*sizeof(uint8_t));
                              ret=write(nodoTx, &mensaje, sizeof(struct can_msg));
                              if(ret!=sizeof(struct can_msg))
                     {err(1, "Failed to send message mode");}
                        display(mensaje);
                          sleep(0.05);
      }
 idmotor=querida->driver;
 Nanterior=idmotor;

while (1)
 {
      idmotor=querida->driver;
      Pg=querida->posicion;
if (Pg!=Panterior)
{   Panterior=Pg;


printf("_____\n");
          printf("%d nodo\n",idmotor);
printf ("angulo deseado en grados\n");
          printf ("%d grados\n",Pg);
          Pui=Pg*4*N*Tr/360;
          conversion(Pui);
  buf7[7]=buf[7];
buf7[6]=buf[6];
buf7[5]=buf[5];
buf7[4]=buf[4];


 Vrpm =querida->velocidad;
          printf ("Velocidad  desea  girar\n");
          printf ("%d revoluciones\n",Vrpm);
          Vui=Vrpm*4*N*Tr*T/60;
          Vui= Vui*65536;

          conversion(Vui);
```

```
            buf8[7]=buf[7];
            buf8[6]=buf[6];
            buf8[5]=buf[5];
            buf8[4]=buf[4];


                    printf("posicion\n");
            mensaje.id=0x600+idmotor;
            memcpy(mensaje.data,buf7,8*sizeof(uint8_t));
              ret=write(nodoTx, &mensaje, sizeof(struct can_msg));
            if(ret!=sizeof(struct can_msg))
            {err(1, "Failed to send message pos");}
            display(mensaje);
            sleep(0.05);

            printf("velociad\n");
            memcpy(mensaje.data,buf8,8*sizeof(uint8_t));
            ret=write(nodoTx, &mensaje, sizeof(struct can_msg));
if(ret!=sizeof(struct can_msg))
              {err(1, "Failed to send message vel");}
            display(mensaje);
            sleep(0.5);

            printf("start\n");
             memcpy(mensaje.data,buf9,8*sizeof(uint8_t));
            ret=write(nodoTx, &mensaje, sizeof(struct can_msg));
            if(ret!=sizeof(struct can_msg))
            {err(1, "Failed to send message start");}
            sleep(1);

            // printf("reset\n");
            memcpy(mensaje.data,buf10,8*sizeof(uint8_t));
            ret=write(nodoTx, &mensaje, sizeof(struct can_msg));
            if(ret!=sizeof(struct can_msg))
            {err(1, "Failed to send message reset");}
            //display(mensaje);
            sleep(0.05);
    }
if ((Pdriver!=Panterior)||(idmotor!=Nanterior))

        {  Pdriver=leer_grados(N,Tr);
           Vdriver=leer_rpm(N,Tr, T);
           Nanterior=idmotor;

           loquehay->posicion=Pdriver;
          loquehay->velocidad=Vdriver;
```

```
//sleep(1);
        }
  }

shmctl(shmid,IPC_RMID,0); //Borrado de la zona de memoria compartida

/*cerrar nodos*/
   close(nodoRx);
   close(nodoTx);

return 0;
}
/**********************librería cabecera.h*******************************/

#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>
#include<string.h>
#include<errno.h>
#include<err.h>
#include<assert.h>
#include"inicializacion.c"
#include"leer.c"
#include"conversion.c"
#include"display.c"
#include"hico_api.h"
#include"shmz.h"/* Fichero que contiene la informaciÃ³n de la estructura de datos a
compartir */
#define BITS 32

//declaracion de funciones
int inicializacion();
int conversion(int valor);
int display(struct can_msg mensaje);
int leer_rpm(int N,int Tr,float T);
int leer_grados(int N,int Tr);
int read_timeout(int fd, struct can_msg *buf, unsignedint timeout);


/**********************función inicialización*********************************/
#include<sys/types.h>
#include<stdio.h>
```

```
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>
#include<string.h>
#include<errno.h>
#include<err.h>
#include<assert.h>
#include"hico_api.h"
#define BITS 32

//variables globales
int nodoRx, nodoTx;
struct can_msg mensaje;
struct can_msg mensaje_rec;


int inicializacion()
{
uint8_t buf0[2] = {0x01,0x01};
uint8_t buf1[8] = {0x2B,0x40,0x60,0x00,0x06,0x00,0x00,0x00};
uint8_t buf2[8] = {0x2B,0x40,0x60,0x00,0x07,0x00,0x00,0x00};
uint8_t buf3[8] = {0x2B,0x40,0x60,0x00,0x0F,0x00,0x00,0x00};

int ret;
int bau=BITRATE_1000k;
int idmotor;

/*abrir nodos*/
nodoTx= open("/dev/can1",O_RDWR);
if(nodoTx<0){err(1, "could not open can nodeTx ");}

/*reset*/
ret=ioctl(nodoTx,IOC_RESET_BOARD);
if(ret!=0){err(1, "could not reset node");}

/*baudios*/
ret=ioctl(nodoTx,IOC_SET_BITRATE,&bau);
if(ret!=0){err(1, "could not set bitrate nodoTx");}

/*start nodos*/
ret=ioctl(nodoTx,IOC_START);
if(ret!=0){err(1, "IOC_START nodoTx");}

memset(&mensaje,0,sizeof(struct can_msg));

//inicializar
```

```
//0
            mensaje.ff=FF_NORMAL;
           mensaje.id=0x00;
           mensaje.dlc=2;
           memcpy(mensaje.data,buf0,2*sizeof(uint8_t));
           ret=write(nodoTx, &mensaje, sizeof(struct can_msg));
if(ret!=sizeof(struct can_msg))
  {err(1, "Failed to send message 1");}
sleep(0.05);


for (idmotor=1;idmotor<3;idmotor++)
{ //1
           mensaje.id=0x600+idmotor;
           mensaje.dlc=8;
           memcpy(mensaje.data,buf1,8*sizeof(uint8_t));
           ret=write(nodoTx, &mensaje, sizeof(struct can_msg));
if(ret!=sizeof(struct can_msg))
  {err(1, "Failed to send message 1");}
sleep(0.05);

           // 2

           memcpy(mensaje.data,buf2,8*sizeof(uint8_t));
           ret=write(nodoTx, &mensaje, sizeof(struct can_msg));
if(ret!=sizeof(struct can_msg))
  {err(1, "Failed to send message 2");}
           sleep(0.05);

           // 3

           memcpy(mensaje.data,buf3,8*sizeof(uint8_t));
           ret=write(nodoTx, &mensaje, sizeof(struct can_msg));
if(ret!=sizeof(struct can_msg))
  {err(1, "Failed to send message 3");}
           sleep(0.05);
 }

return 0;
}

/*****************************función conversion*************************/
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
```

```
#include<fcntl.h>
#include<string.h>
#include<errno.h>
#include<err.h>
#include<assert.h>
#include"hico_api.h"
#define BITS 32


uint8_t buf[8] = {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};

int conversion(int valor)
{
int ret,i,j;
int exp,flag,caso;
int binNum[BITS];
int hex[8] = {0,0,0,0,0,0,0,0};


flag=0;
if (valor<0)
  {flag=1;
valor=-valor-1;
}
for (i=0;i<BITS; i++)
{binNum[i]=flag^(valor%2);
valor /=2;
  }
printf("\n----------numero en binario----------\n");
for (i=BITS-1;i>=0;i--)
{printf("%d ",binNum[i]);
  }

printf("\n----------numero en hexadecimal----------\n");
for (i=0,j=0;i<BITS;i=i+4,j++)
{ caso=binNum[i+3]*1000+binNum[i+2]*100+binNum[i+1]*10+binNum[i];
switch (caso)
    {    case 0:
//    printf("caso0 \n ");
hex[j]= 0;
//    printf("Decimal:%d\n",hex[j]);
          //   printf("Hex:%x\n",hex[j]);
          break;
                case 01:
                //         printf("caso 1 \n ");
hex[j]= 1;
```

```
//   printf("Decimal:%d\n",hex[j]);
          //   printf("Hex:%x\n",hex[j]);
          break;
          case 10:
          // printf("caso 2 \n ");
hex[j]= 2;
//   printf("Decimal:%d\n",hex[j]);
          //   printf("Hex:%x\n",hex[j]);
          break;
                        case 11:
                        //  printf("caso 3 \n ");
hex[j]= 3;
//   printf("Decimal:%d\n",hex[j]);
          //   printf("Hex:%x\n",hex[j]);
          break;
                        case 100:
                        // printf("caso 4 \n ");
hex[j]= 4;
//   printf("Decimal:%d\n",hex[j]);
          //   printf("Hex:%x\n",hex[j]);
          break;
                        case 101:
                  //        printf("caso 5 \n ");
hex[j]= 5;
//   printf("Decimal:%d\n",hex[j]);
          //   printf("Hex:%x\n",hex[j]);
          break;
          case 110:
          //  printf("caso 6 \n ");
hex[j]= 6;
//   printf("Decimal:%d\n",hex[j]);
          //   printf("Hex:%x\n",hex[j]);
          break;
          case 111:
          // printf("caso 7 \n ");
hex[j]= 7;
//   printf("Decimal:%d\n",hex[j]);
          //   printf("Hex:%x\n",hex[j]);
          break;
          case 1000:
          //  printf("caso 8 \n ");
hex[j]= 8;
//   printf("Decimal:%d\n",hex[j]);
          //   printf("Hex:%x\n",hex[j]);
          break;
          case 1001:
```

```
            //printf("caso 9 \n ");
hex[j]= 9;
//   printf("Decimal:%d\n",hex[j]);
            //   printf("Hex:%x\n",hex[j]);
            break;
            case 1010:
            // printf("caso A \n ");
hex[j]= 10;
//   printf("Decimal:%d\n",hex[j]);
            //   printf("Hex:%x\n",hex[j]);
            break;
            case 1011:
            //  printf("caso B \n ");
hex[j]= 11;
//   printf("Decimal:%d\n",hex[j]);
            //  printf("Hex:%x\n",hex[j]);
            break;
            case 1100:
            //  printf("caso C \n ");
hex[j]= 12;
//   printf("Decimal:%d\n",hex[j]);
            //   printf("Hex:%x\n",hex[j]);
            break;
            case 1101:
            //  printf("caso D \n ");
hex[j]= 13;
//    printf("Decimal:%d\n",hex[j]);
            //   printf("Hex:%x\n",hex[j]);
            break;
            case 1110:
            // printf("caso E \n ");
hex[j]= 14;
//   printf("Decimal:%d\n",hex[j]);
            //   printf("Hex:%x\n",hex[j]);
            break;
            case 1111:
            // printf("caso F \n ");
hex[j]= 15;
//   printf("Decimal:%d\n",hex[j]);
            //   printf("Hex:%x\n",hex[j]);
            break;
            default:
            printf("mal");
            break;
              }
  }
```

```
buf[7]= hex[7]*16 +hex[6];
buf[6]= hex[5]*16 +hex[4];
buf[5]= hex[3]*16 +hex[2];
buf[4]= hex[1]*16 +hex[0];

//printf("_____\n");
for (i=4;i<8; i++){
        printf("%x ",buf[i]);
}

printf("conversion hecha\n");
return 0;
}

/**********************************función leer*****************************/
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>
#include<string.h>
#include<errno.h>
#include<err.h>
#include<assert.h>
#include"hico_api.h"
#define BITS 32

//variables globales
int nodoRx, nodoTx;
struct can_msg mensaje;
struct can_msg mensaje_rec;
int idmotor;



int leer_rpm(int N,int Tr,float T)
{
int valor,ret;
        float revoluciones;
        int i=1;
uint8_t buf14[8]= {0x40,0x6C,0x60,0x00,0x00,0x00,0x00,0x00}; //velocidad actual

sleep(1); //hace cosas raras si no esta puesto (lo de para un lado y luego para el otro)
//printf("Valor velocidad según driver \n");
```

```
   mensaje.id=0x600+idmotor;
   mensaje.dlc=8;
memcpy(mensaje.data,buf14,8*sizeof(uint8_t));
ret=write(nodoTx, &mensaje, sizeof(struct can_msg));
if(ret!=sizeof(struct can_msg))
    {err(1, "Failed to recive message vel");}
//display(mensaje);
sleep(0.05);

while(i== 1)
     {
          ret=read_timeout(nodoTx,&mensaje_rec,2000);
if(ret==0)
       {errx(1,"timeout - could not read the message. Check the cable!");}
                    if (mensaje_rec.data[1]== 108)
                    { if (mensaje_rec.data[2]== 96)
                          i=0;
                    }
sleep(0.1);
                    }
// display(mensaje_rec);

valor=mensaje_rec.data[7]*16777216 + mensaje_rec.data[6]*65536
+mensaje_rec.data[5]*256 + mensaje_rec.data[4];
valor = valor/65536;
revoluciones= (valor*60)/(4*N*Tr*T);
// printf("Revoluciones %f\n",revoluciones);

return revoluciones;
}

int leer_grados(int N,int Tr)
{
int valor,ret;
          float grados;
          int i=1;
uint8_t buf12[8]={0x40,0x64,0x60,0x00,0x00,0x00,0x00,0x00}; //chequea el valor de la
posicion actual del motor

sleep(1);
// printf("Valor posicion segun driver \n");
   mensaje.id=0x600+idmotor;
   mensaje.dlc=8;
memcpy(mensaje.data,buf12,8*sizeof(uint8_t));
ret=write(nodoTx, &mensaje, sizeof(struct can_msg));
if(ret!=sizeof(struct can_msg))
```

```
    {err(1, "Failed to recive message vel");}
//display(mensaje);
sleep(0.05);

while(i== 1)
     {
          ret=read_timeout(nodoTx,&mensaje_rec,2000);
if(ret==0)
       {errx(1,"timeout - could not read the message. Check the cable!");}
                    if (mensaje_rec.data[1]== 100)
                    { if (mensaje_rec.data[2]== 96)
                          i=0;
                    }
sleep(0.1);
                    }
//  display(mensaje_rec);

valor=mensaje_rec.data[7]*16777216 + mensaje_rec.data[6]*65536
+mensaje_rec.data[5]*256 + mensaje_rec.data[4];
grados= (valor*360)/(4*N*Tr);
// printf("Grados %f\n",grados);


return grados;
}

int read_timeout(int fd, struct can_msg *buf, unsignedint timeout)
{
fd_set fds;
struct timeval tv;
int sec,ret;
   FD_ZERO(&fds);

sec=timeout/1000;
   tv.tv_sec=sec;
   tv.tv_usec=(timeout-(sec*1000))*1000;

   FD_SET(fd,&fds);

ret=select(fd+1,&fds,0,0,&tv);
if(ret==0){
          return 0; /* timeout */
   } elseif (ret<0) {
          return errno;
   } else {
          assert(FD_ISSET(fd,&fds));
```

```
        ret=read(fd,buf,sizeof(struct can_msg));
        return ret;
    }
}

/**********************función display*****************************/
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>
#include<string.h>
#include<errno.h>
#include<err.h>
#include<assert.h>
#include"hico_api.h"
#define BITS 32
int nodoRx, nodoTx;
struct can_msg mensaje;

int display(struct can_msg mensaje)
{
        int i;
        printf("\tID: %X\n",mensaje.id);
        //printf("\tnodo: %X\n",mensaje.node);
        printf("\tData: ");
        for(i=0; i < mensaje.dlc; i++){
                printf("%X - ",mensaje.data[i]);
        }
        printf("\n");
return 0;
}


/**********************función matlaver*********************************/
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include"shmz.h"/* Fichero que contiene la informaciÃ³n de la estructura de datos a
compartir */
#include"mex.h"
#include"matrix.h"
```

```
void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[]) {

int shmid;

double *v,*z;
double y, x;

int vquerida,pquerida,nodo;

ver *loquehay;


/* CreaciÃ³n de la zona de memoria compartida */
if((shmid = shmget(CLAVE_ver, sizeof(ver), IPC_CREAT|0666)) == -1){
perror("shmget");
exit(EXIT_FAILURE);
}

/* ObtenciÃ³n del puntero a la estructura de datos compartida */

loquehay = (ver *)shmat(shmid,0,0);

/*xtimesy.c es de hay de donde lo he sacado*/
x=loquehay->velocidad; /*x=525;*/
plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
z = mxGetPr(plhs[0]);
*(z) = x;

y=loquehay->posicion;/*y=254;*/
plhs[1] = mxCreateDoubleMatrix(1,1, mxREAL);
v = mxGetPr(plhs[1]);
*(v) = y;

}

/*****************************funciónmatlaenlace************************/
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include"shmz.h"/* Fichero que contiene la informaciÃ³n de la estructura de datos a
compartir */
#include"mex.h"
#include"matrix.h"
```

```
void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[]) {

int shmid;

int vquerida,pquerida,nquerida;
memo *querida;


/* CreaciÃ³n de la zona de memoria compartida */
if((shmid = shmget(CLAVE_SHM, sizeof(memo), IPC_CREAT|0666)) == -1){
perror("shmget");
exit(EXIT_FAILURE);
}


/* ObtenciÃ³n del puntero a la estructura de datos compartida */
querida = (memo *)shmat(shmid,0,0);

vquerida = (int)mxGetScalar(prhs[0]);
pquerida = (int)mxGetScalar(prhs[1]);
nquerida = (int)mxGetScalar(prhs[2]);
querida->velocidad = vquerida;
querida->posicion = pquerida;
querida->driver = nquerida;


mexPrintf("Valores que yo mando:vel %d, pos%d, nodo %d\n", vquerida, pquerida,
nquerida);
/*  mexPrintf("velocidad que quiero:%d, posicion que quiero:%d\n", vquerida, pquerida);*/

}

/*****************************función matlanodo************************/
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include"shmz.h"/* Fichero que contiene la informaciÃ³n de la estructura de datos a
compartir */
#include"mex.h"
#include"matrix.h"

void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[]) {
```

```
int shmid;

int nquerida;
memo *querida;


/* CreaciÃ³n de la zona de memoria compartida */
if((shmid = shmget(CLAVE_SHM, sizeof(memo), IPC_CREAT|0666)) == -1){
        perror("shmget");
        exit(EXIT_FAILURE);
}


/* ObtenciÃ³n del puntero a la estructura de datos compartida */
querida = (memo *)shmat(shmid,0,0);

    nquerida = (int)mxGetScalar(prhs[0]);
    querida->driver = nquerida;


     mexPrintf("Valores que yo mando: nodo %d\n", nquerida);
/*   mexPrintf("velocidad que quiero:%d, posicion que quiero:%d\n", vquerida, pquerida);*/

}
/************************************libreria shmz*******************/
/* Fichero shmz.h, contiene informaciÃ³n sobre la zona de memoria que se
pretendecompartir entre diversos procesos, será preciso incluirlo en la cabecera del fichero
deaquellos procesos que se conecten a la zona de memoria común */

/* Definición de la clave de acceso a la zona de memoria común */
#define CLAVE_SHM ((key_t) 1001)
#define CLAVE_ver ((key_t) 1002)/*esto lo he puesto yo*/
/* Estructura de datos que se pretende compartir en la zona de memoria común */
typedefstruct{
int posicion;
int velocidad;
int driver
}memo;

typedefstruct{
int posicion;
int velocidad;
       }ver
```

```
/**************************interfanfin.m*******************************/
function varargout = interfazfin(varargin)

gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
'gui_Singleton',  gui_Singleton, ...
'gui_OpeningFcn', @interfazfin_OpeningFcn, ...
'gui_OutputFcn',  @interfazfin_OutputFcn, ...
'gui_LayoutFcn',  [] , ...
'gui_Callback',   []);
if nargin && ischar(varargin{1})
   gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
   [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
   gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before interfazfin is made visible.
function interfazfin_OpeningFcn(hObject, eventdata, handles, varargin)
handles.output = hObject;

humanoide =imread('entero.jpg');
axes(handles.humanoide);
image(humanoide);
axis off;

logo =imread('logo2.jpg');
axes(handles.logo);
image(logo);
axis off;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes interfazfin wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = interfazfin_OutputFcn(hObject, eventdata, handles)
varargout{1} = handles.output;
```

```
% --- Executes on button press in tobderecho.
function tobderecho_Callback(hObject, eventdata, handles)
tob_derecho


% --- Executes on button press in tobizquierdo.
function tobizquierdo_Callback(hObject, eventdata, handles)
tob_izquierdo

% --- Executes on button press in rodderecha.
function rodderecha_Callback(hObject, eventdata, handles)
rod_derecha

% --- Executes on button press in rodizquierda.
function rodizquierda_Callback(hObject, eventdata, handles)
rod_izquierda

% --- Executes on button press in cadera.
function cadera_Callback(hObject, eventdata, handles)
cadera


% --- Executes on button press in pierderecha.
function pierderecha_Callback(hObject, eventdata, handles)
pierderecha

% --- Executes on button press in pierizquierda.
function pierizquierda_Callback(hObject, eventdata, handles)
pierizquierda

% --- Executes on button press in homderecho.
function homderecho_Callback(hObject, eventdata, handles)
homderecho

% --- Executes on button press in cododerecho.
function cododerecho_Callback(hObject, eventdata, handles)
cododerecho

% --- Executes on button press in manoderecha.
function manoderecha_Callback(hObject, eventdata, handles)
manoderecha

% --- Executes on button press in homizquierdo.
function homizquierdo_Callback(hObject, eventdata, handles)
```

```
homizquierdo

% --- Executes on button press in codoizquierdo.
function codoizquierdo_Callback(hObject, eventdata, handles)
codoizquierdo

% --- Executes on button press in manoizquierda.
function manoizquierda_Callback(hObject, eventdata, handles)
manoizquierda




/**********************************tob_derecho**************************/
function varargout = tob_derecho(varargin)
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
'gui_Singleton',  gui_Singleton, ...
'gui_OpeningFcn', @tob_derecho_OpeningFcn, ...
'gui_OutputFcn',  @tob_derecho_OutputFcn, ...
'gui_LayoutFcn',  [] , ...
'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before tob_derecho is made visible.
function tob_derecho_OpeningFcn(hObject, eventdata, handles, varargin)

tobillo=imread('tobillo_derecho.jpg');
axes(handles.tobillo);
image(tobillo);
axis off

logo =imread('logo2.jpg');
axes(handles.logo);
image(logo);
```

```
axis off;


% Choose default command line output for tob_derecho
handles.output = hObject;
%handles.driver=1;
% Update handles structure
guidata(hObject, handles);

initialize_gui(hObject, handles, false);



% --- Outputs from this function are returned to the command line.
function varargout = tob_derecho_OutputFcn(hObject, eventdata, handles)
varargout{1} = handles.output;




function velocidad_Callback(hObject, eventdata, handles)
velocidad= str2double(get(handles.velocidad, 'String'));
if isnan(velocidad)
   set(hObject, 'String', 0);
errordlg('Introduce un nÃºmero','Error');
end

guidata(hObject,handles)


% --- Executes during object creation, after setting all properties.
function velocidad_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end




function posicion_Callback(hObject, eventdata, handles)
posicion= str2double(get(handles.posicion, 'String'));
if isnan(posicion)
   set(hObject, 'String', 0);
errordlg('Introduce un nÃºmero','Error');
```

```
end
if posicion>20
   set(hObject, 'String', 0);
errordlg('Introduce un Ã¡ngulo menor de 20 grados','Error');
end
if posicion<-20
   set(hObject, 'String', 0);
errordlg('Introduce un Ã¡ngulo mayor de -20 grados','Error');
end


% --- Executes during object creation, after setting all properties.
function posicion_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end



function initialize_gui(fig_handle, handles, isreset)
set(handles.uipanel2, 'SelectedObject', handles.movsagital);
set(handles.text5, 'String','1')

function vertiempo_gui(fig_handle, handles, isreset)
while 1
[vel,pos]=matlaver;
set(handles.text3, 'String', num2str(vel));
set(handles.text4, 'String', num2str(pos));
pause(0.5);
end;


% --- Executes on button press in posicionar.
function posicionar_Callback(hObject, eventdata, handles)
vel = str2double(get(handles.velocidad, 'String'));
pos = str2double(get(handles.posicion, 'String'));
nodo = str2double(get(handles.text5, 'String'));
matlaenlace(vel,pos,nodo)



% --- Executes on button press in ver.
```

```
function ver_Callback(hObject, eventdata, handles)
[vel,pos]=matlaver
set(handles.text3, 'String', num2str(vel))
set(handles.text4, 'String', num2str(pos))

% --- Executes when selected object is changed in uipanel2.
function uipanel2_SelectionChangeFcn(hObject, eventdata, handles)
if (hObject == handles.movsagital)
   set(handles.text5, 'String', '1');
else
   set(handles.text5, 'String', '2');
end
nodo = str2double(get(handles.text5, 'String'));
matlanodo(nodo)



function edit3_Callback(hObject, eventdata, handles)


% --- Executes during object creation, after setting all properties.
function edit3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end
```